NO.33

编程狂人

Programming Madman

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:http://www.tuicool.com/mags/53c3966ed91b14216e19aaed

欢迎下载推酷客户端体验更多阅读乐趣

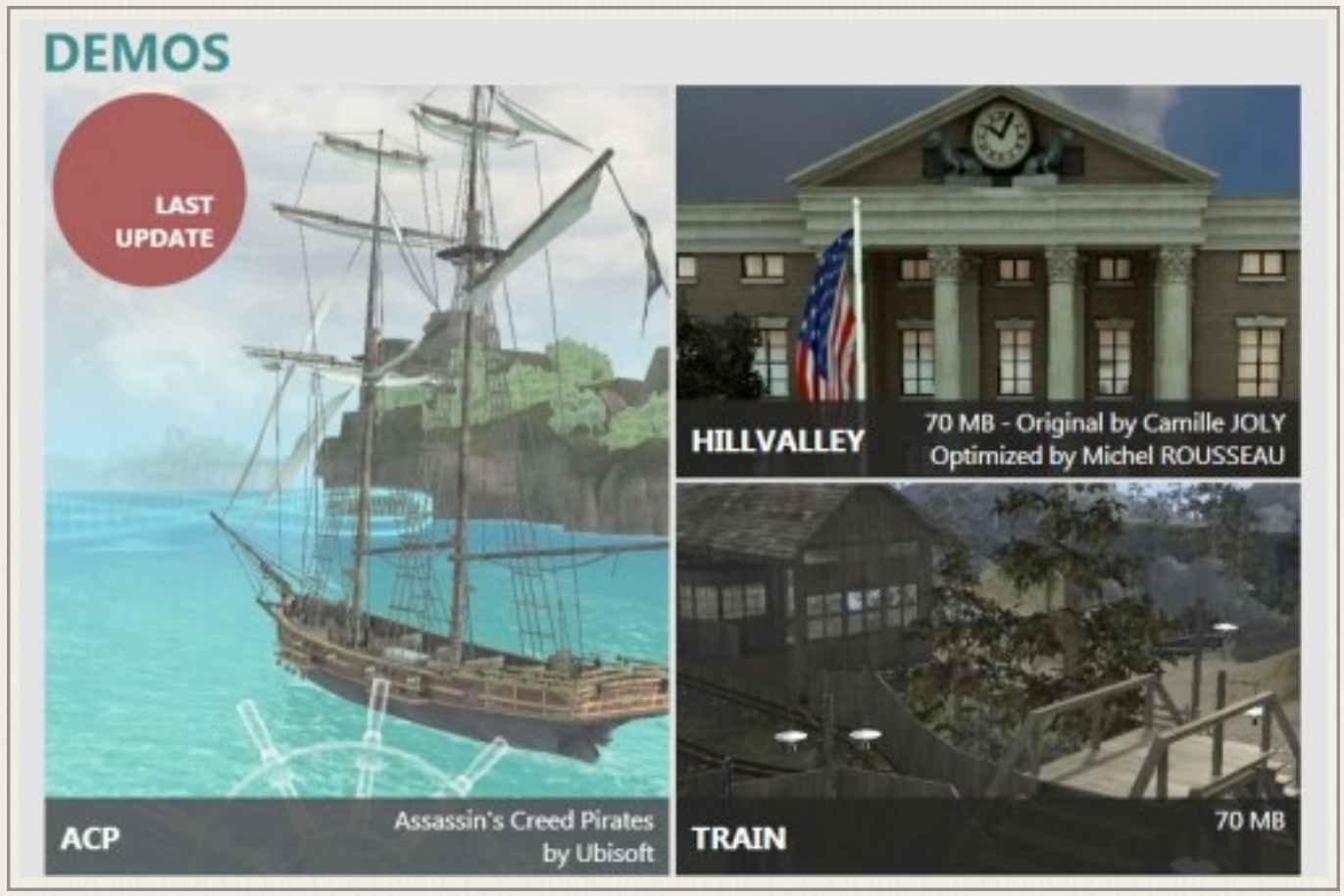
目录

Babylon.js：拥有微软背景的开源3D游戏引擎

作者:唐小引

摘要：Babylon.js是一款基于WebGL、HTML5和JavaScript的开源3D游戏引擎，由微软员工David Catuhe主导开发。配合Babylon.js，开发者可以更方便快捷地完成光线、轮船纹理、海浪等的3D建模，从而带来最佳的呈现效果。

Babylon.js是一款基于WebGL、HTML5和JavaScript的开源3D游戏引擎，由微软员工David Catuhe主导开发，其团队主要包括：开发者David Rousset、Pierre Lagarde，以及3D美工Michel Rousseau。配合Babylon.js，开发者可以更好地利用WebGL技术，更方便快捷地完成光线、轮船纹理、海浪等的3D建模，从而带来最佳的 呈现效果。



在Babylon.js官网上，拥有着众多非常酷炫的Demo，包括此前微软与育碧联合推出的“刺客信条·海盗”Demo。几经更新与完善之 后，Babylon.js已

更新至1.12版本，相比之前的版本，除了不断的Bug修复之外，Babylon.js还新增了许多非常牛叉的新特性，并对已 有功能进行了完善。

主要更新如下：

- 完全支持TypeScript开发；
- 物理插件：支持开发者向Babylon.js添加自有的物理引擎；
- BABYLON.Action：开发者可以创建复杂的交互系统；
- 几何系统支持；
- 新增基于Vincent Thibault的TGA纹理支持；
- 新增BABYLON.Gamepads和BABYLON.Gamepad，开启Gamepad API，实现掌机游戏体验；
- 新增VertexData.CreateGroundFromHeightMap、Tools.CreateScreenshot函数；
- 修复沙盒工具Bug；
- 为与Mesh.updateVerticesData保持一致，Mesh.setVerticesData签名更改为(kind, values, updatable)；
- 更多信息，可查看Babylon.js Releases Notes。

现在，您还可以进入Babylon.js的mobilehub主页进行资源分享和讨论。开发者可登陆GitHub下载Babylon.js最新版本，想要了解更多关于Babylon.js的信息，可直接查看Babylon.js官网。

原文链接: http://www.csdn.net/article/2014-07-07/2820558-babylonjs

淘宝前端工程师：国内WEB前端开发十日谈

作者: 一色

一直想写这篇“十日谈”，聊聊我对Web前端开发的体会，顺便解答下周围不少人的困惑和迷惘。我不打算聊太多技术，我想，通过技术的历练，得到的反思应当更重要。

我一直认为自己是“初级”前端开发工程师，一方面我入道尚浅，只有短短几年，另一方面我自知对技术的钻研并不深入，可能是由于环境的原因，当然 最重要的是，我幸运的参与到互联网崛起的浪潮之巅。时势造就了一批技能薄弱但备受追捧的“弄潮者”，这在很大程度上影响我们对“技术本质”的洞察力，多年 来也一直未有成体系的“前端技术”布道佳作，以至于当下多数人对前端技术的了解，盖始于表述并不严谨的岗位招聘描述，而这正恰恰反映了Web前端开发对自 身的模糊定位。对于很多Web前端工程师来说，初尝禁果的快感无法持续很久，就陷入一轮又一轮的迷惘，思索自己的职业规划，试图寻找到适合自己的成长道 路、看清自身技能的瓶颈，寻找突破。但遗憾的是，Web前端技术被广泛接纳时日尚短，没有多少励志的成功样板可供遵循。然而情况不总是这么糟，毕竟Web 前端技术是一门“技术”，和计算机科学系出同门，只是因为互联网的高速崛起而被蒙上了迷雾，遮住了双眼，让我们傻傻看不清时局。

那么，如何定义Web前端技术岗位边界？Web前端技术的价值体现在何处？前端工程师的价值仅仅体现在物以稀为贵吗？前端工程师的初级、中级、 高级和专家之间到底如何界定？当前“我”处在什么位置？接下来的路子应当怎样走？何谓前端技术之“道”？我想多数人都思考过这些问题，本篇“十日谈”里的 观点可能有些偏激，但抛砖引玉，读者权且把这些言论当作一个引子吧。

第一日：初尝禁果

【上帝说：“要有光！”便有了光】

万物生灵、阳光雨露盖源于造物之初的天工开物，我们无法想象上帝创造光明之前的世界模样。但幸运的是，前端开发没有神祇般的诡魅。这个技术工种 的孕育、定型、发展自有轨迹，也颇有渊源，当然，这非常容易理解。不严格的讲，在杨致远和费罗在斯坦福大学的机房里擂掇出Yahoo!时，Web前端技术 就已经开始进入公众视野，只不过当时没有一个响亮的名字。从那时起，“基于浏览器端的开发”就成了软件开发的新的分支，这也是Web前端技术的核心，即不 论何时何地何种系统以及怎样的设备，但凡基于浏览器，都是Web前端开发的范畴（当然，这个定义很狭隘，下文会提到）。

在2000年之后浏览器技术渐渐成熟，Web产品也越来越丰富，中国有大批年轻人开始接触互联网，有一点需要注意，大部分人接触互联网不是始于 对浏览器功能的好奇，而是被浏览器窗口内的丰富内容所吸引，我们的思维模式从一开始就被限制在一个小窗口之内，以至于很长时间内我们将“视觉”认为是一种 “功能”，Web产品无非是用来展现信息之用。起初的入行者无一例外对“视觉”的关注超过了对“内容”的重视，先让页面看起来漂亮，去关注 html/css，沿着“视觉呈现”的思路，继续深入下去。因此，这类人是被“视觉”所吸引，从切页面入行，着迷于结构化的html和书写工整的css， 喜欢简洁优雅的UI和工整的页面设计，之后开始接触视觉特效，并使用jQuery来实现视觉特效，以此为线索，开始深入研究Dom、Bom和浏览器的渲染 机制等，html/
css在这些人手中就像进攻兵器，而JavaScript则更如防守的盾牌。

还有另外一群人从另一条道路接触Web前端，即工程师转行做前端，他们有较多的后台语言开发背景，从读写数据开始，渐渐触及浏览器端，接触 JavaScript库，起初是在html代码上加js逻辑，后来开始涉及html和css，他们喜欢OO、逻辑清晰、结构悦目的代码，更关注界面背后的 “程序语言”和数据逻辑。html/
css在这些人手中则更像盾牌，而JavaScript更如进攻的兵器。

应当说这两类人是互补的，他们各自了解浏览器本质的一部分，一拨人对渲染引擎了如指掌，另一拨人则将JS引擎奉为至宝，其实任何一部分的优势发 挥出来都能做出精品。大部分前端工程师都能从这两条渊源中找到自己的影子。但，这两类人的思维模式和观点是如此不同，以至于形成了一些不必要的对抗，比如 在某些公司，干脆将Web前端技术一分为

二，“切页面的”和“写js的”。这样做看上去明确了分工提高了效率，但他对员工的职业发展带来巨大伤害。在第二 日“科班秀才”中会有进一步讨论。

我应该属于第二类，即在学校正儿八经的学习C/Java和C#之类，以为大学毕业后能去做ERP软件、桌面软件或者进某些通信公司写TCP /IP相关的程序。校园招聘时选择了中国雅虎，因为当年（08年）雅虎还是有一点儿名气，而且我听说雅虎比较算技术流的公司……自此就上了贼船，一发不可 收拾。

在雅虎的这段时间，我有幸接触到一股正气凛然的技术流派，也形成了我对前端技术的一些基本看法，这些基本观点一直影响我至今。

【优雅的学院派】

当年雅虎的技术流派正如日中天，拥有众多“之父”级的高人，所营造出的Hack氛围实在让人陶醉的无法自拔，那段时间我甚至宁愿加班到深夜阅读 海量的文档和源代码，感觉真的很舒服，我深深的被雅虎工程师这种低调务实、精工细琢的“服务精神”所打动，而这种不起眼的优秀品质很大程度的影响雅虎产品 的用户体验和高质量的技术输出。那么，何谓“服务精神”？即你所做的东西是服务于人的，要么是产品客户、要么是接手你项目的人、要么是使用你开发的功能的 人，所以技术文档成为伴随代码的标配。因此，工程师之间通过代码就能做到心有灵犀的沟通。这是工程师的一项基本素质，即，思路清晰的完成项目，且配备了有 价值的技术文档，如果你的程序是给其他程序员用的，则更要如此，就好比你制造一款家电都要配备说明书一样。因此，YDN成了当时最受全球程序员最喜爱的技 术文档库，这种优雅务实的“学院气息”让人感觉独具魅力。

让人感觉奇怪的是，在中文社区始终未见这种学院派。甚至在具有先天开源优势的Web前端技术社区里也是波澜不惊，可见写一篇好的技术文案真的比 登天还难。我所见到的大部分所谓文档索性把代码里输出数据的语句块拷贝粘贴出来，至于为什么数据格式要设计成这样、如果字段有修改怎么做、编码解码要求如 何等等关键信息只字不提，或者开发者也没想过这些问题呢。因此，我们一直在强调代码的质量和可维护性，但一直以来都未见效，盖源于缺少这种“服务”意识的 灌输。这种意识在下文中还会多次提到，因为它能影响你做事的每个细节，是最应当首先突破的思想纠结。

除了意识问题，另一方面是技术问题，即文笔。这也是工程师最瞧不上眼的问题，难以置信这竟然是阻碍工程师突破瓶颈的关键所在。我已看到过数不清 的人在晋升这道关卡吃了大亏，很多工程师技术实力很强，但就是表达不出来，要么罗列一大堆信息毫无重点、要么毫无趣味的讲代码细节，不知云云。除非你走狗 屎运碰到一个懂技术的老板，否则真的没办法逃脱码农的宿命。但大部分人还振振有词不以为然。而在Web前端开发领域情况更甚。前端工程师是最喜欢搞重构 的，但在快节奏的需求面前，你很难用“提高了可维护性”、“提升了性能”这类虚无缥缈的词藻为自己争取到时间来搞重构，说的露骨一点，可能你真的对某次重 构带来的实际价值无法量化，只是“感觉代码更整洁了”而已。我会在下文的“伪架构”中会展开分析前端工程师的这种浮躁献媚的技术情结。而这正是前端工程师 最欠缺的素质之一：用数据说话，用严谨科学的论据来支撑你的观点，老板不傻，有价值的东西当然会让你去做。

当然，情况不总是这么糟糕，我们看到中文社区中已经锻炼出了很多写手，他们在用高质量的文字推销自己的技术理念，这是一个好兆头，好的文笔是可 以锻炼出来的。而在职场，特别是对前端工程师这个特殊职位来讲，这种基本技能可以帮你反思梳理需求的轻重缓急，从凌乱的需求中把握七寸所在。因为当你开始 认真写一封邮件的时候，这种思考已经包含其中了。

所以，雅虎技术的推销是相对成功和远播的。关键在于两方面，扎实的技术功底和高超的写手。而真正的技术大牛一定是集两者与一身，不仅钻研剑道，还能产出秘籍。这也是Yahoo!优雅的学院派气息的动力源泉。国内很多技术团体想在这方面有所建树，应当首先想清楚这一点。

【规范的破与立 1】

雅虎的技术运作非常规范，刚才已经提到，包括技术、组织、文化，一切看起来有模有样，也堪称标杆，自然成了国内很多技术团队和社区的效仿对象。一时间各种“规范“成风、各色“标准“大行其道，结果是质量参差不齐。

我们到底需要什么样的规范？雅虎的技术规范到底有何种魔力？以何种思路构建的规范才是货真价实的？规范有着怎样的生命周期？想清楚这

些问题，能很大程度减轻很多Web前端工程师的思想负担，看清一部分技术本质，避免盲目跟风。

我们的确需要规范，但好的规范一定是务实的，一定是“解决问题“的。比如针对项目构建的DPL可以收纳公用的视觉元件以减少重复开发、规定某 OPOA项目的事件分发原则以确立增量开发的代码惯性。反之，糟糕的规范却显得过于“抽象“，比如页面性能指标、响应式设计原则。另外，尽管他山之石可以 攻玉，但拿来主义有一个大前提，就是你了解你的项目的关键问题，你要优先解决的是些关键问题，而外来规范正好能解决你的问题。因此规范是一本案头手册，是 一揽子问题的解决方案，应当是“字典”，而不是“教程“。可见规范的源头是“问题”。所以，当你想用CoffeeScript重构你的项目时、当你想引入 CommonJS规范时、当你想在页面中揉进Bootstrap时、当你打算重复造轮子搞一套JS库时、当你想重写一套assets打包工具时，想想这些 东东解决了你的什么问题？会不会带来新的问题、把事情搞复杂了？还是为了尝鲜？或者为了在简历中堂而皇之的写上使用并精通各种新技术?

规范之立应当有动因，动因来源于项目需求，项目需求则来自对产品的理解和把握，这是Web前端初级工程师走向中级甚至高级的一次重要蜕变，软件 工程领域早就有“架构师”角色，而架构师往往存在于项目需求分析和概设、详设阶段。我看到的情况是，Web前端工程师的思维过多的限制在“界面”之内，向 前和产品需求离的太远（认为这是视觉设计师的事）、向后和数据逻辑又隔离开来（认为这是后台工程师该干的事），因此前端规范也大都泛泛，无关项目痛痒，成 了玩具。

雅虎技术规范的优秀之初在于它们解决问题。所以，学习使用规范应当多问一句，“他们为什么这样做？”其实，想清楚这些问题时，脑海中自然形成了一种“遇山开山”的创造性思维。

【规范的破与立 2】

如果说新技术的尝鲜缺少针对性，但至少满足程序员的某种洁癖和快感，那么“负担”从何而来呢？对于初学者来说，有价值学习资料可能只有这些规范，如果说规范价值不大，那又当从何入手呢?

刚才我说的不是依赖于规范，而是对规范的反思，摆脱规范灌输给我们的思维定势。新人们大概是看了Wiki中的很多指标、结论、实践，在做

项目之 初就附加了不少“八股式”的负担，甚至影响我们对项目关键需求和关键问题的洞察力和判断力，负担过重就无法轻装上阵，Wiki中提到的这些指标和规范是结 论性的，是大量的实践之后得出的，也只有经历过大量实践才会真正理解这些结论，比如DomReady时间和http请求数是否有因果关系，http请求数 增加是否真的会导致页面性能下降，什么条件下会导致性能下降？我们从那些条文和结论中无法找到答案。

举个具体的例子，Kissy刚刚出了DPL，也是一大堆结论，比如他的布局就采用了经典的双飞翼，使用容器浮动来实现，那么，这种做法就是不可 撼动的“标准”吗？看看淘宝车险首页，布局容器齐刷刷的inline-block，只要顶层容器去掉宽度，布局容器自身就能根据浏览器宽度调整自然水平/ 垂直排列，轻易的适应终端宽度了。

再比如，淘宝旅行计划项目中的部署方式，也没有完全使用Loader管理依赖，而是将依赖层级做的很少，业务逻辑使用脚本来合并，这样就可以更容易在build环节加入语法检查和代码风格检查。

类似这种摆脱原有编程思维，有针对性的用新思路新方法解决问题的做法显然让人感觉更加清爽，编程的乐趣也正体现在打破常规的快感之中，小马曾经 说过：“制造规范是为了打破规范”，万不要因为这些规范标准加重负担，导致开始做一个简单页面时也显得缩手缩脚，无法放开身手。大胆的动手实践，才能真正 得出属于自己的“结论 “和“标准“，才会真正深刻理解那些“结论”的意义所在。代码写的多了，自然熟能生巧，也容易形成成熟的技术观点。

在这个过程中，我们唯一的对手是懒惰，惰于思考，就无法真正发现问题，自然形不成自己的观点。还是那句话，任何规范、方法、结论、实践都是为了 解决项目中的问题的，所以，我们所接触到那些看似“八股文”式的规范标准也是为了解决某些问题而提出的，想清楚这些问题，理解方法论背后的“因“，内心自 然有“果”。

因此，“着眼当下、对症下药”的品质就显得弥足珍贵了，比如，双飞翼布局方法是为了解决一套（html）代码适应多种布局设计，这里的布局相对 于固定的产品来说也是固定的，而无针对终端的自适应（适用于移动端的榻榻米布局似乎还没有最佳实践）。这是双飞翼产生的背景，如今终端环

境较之5年前已经 翻天覆地，问题早已不在“多种布局”上，而在“终端适应“上，这才是我们面临的问题，需要我们给出新的技术方案。

所以，勤于思考，轻装上阵，大胆实践，勇于创新，发掘问题所在，实打实的解决（潜在）问题，这才是我们真正需要的能力。放下思维定势枷锁，也会有一种豁然开朗的感觉。

第二日：科班秀才

【秀才仕途】

Web前端工程师是一个特别的岗位，只存在于互联网领域。最近几年随着互联网产业的火爆，对前端工程师的需求量暴增，兵源几近枯竭。各大公司技术掌门一定都有过类似的苦恼：“招一个靠谱的前端工程师、难于上青天”。

我想，一部分原因是，当前不少入道的前端工程师大都是转行而来，毕竟，正儿八经的学校里也不会教这玩意，觉得“切页面”有啥好教的，甚至不觉得 html/css是一门语言。转行这事自不必详说，大家也各自瞄准当前市场需求，造成的现象是，初级前端工程师堆成山，中高级人才却一将难求，计算机系的 科班出身就更加凤毛麟角了。一方面反映了教育部门的后知后觉，另一方面也体现了大部分人急功近利的跟风。当然最重要的原因是，所谓中国“第一代前端工程 师”并未做好布道的工作。导致大家对于基础和潜力的态度从之前的忽视演变为如今的蔑视。所谓基础，就是在大学上的那些计算机基础课。所谓潜力，就是戒骄戒 躁的务实作风。这些会在后文中多次提到。

对于科班出身的莘莘学苗来说，根正苗红本身就是一种优势，事实证明，这些人在前端技术上的成长轨迹有一定的套路，而且大都能如期的突破技能瓶颈。从一个人大学毕业到他最满意的工作状态，中间会经过几个阶段。

前2年是学习技能的阶段，这个阶段主要精力放在专业技能的提升上，2年内起码要赶上平均水平，即所谓“中级“，在这个阶段的人通常对软技能不怎 么关注，沟通能力达不到平均水平，基本上是来啥活干啥活，干不完就加班的这种，对需求的合理性不甚理解，对项目也没什么把控，尽管在技能上有提高的空间， 也不是公司最需要的人，但有不少成长空间。

工作2-3年的人在前端技能上趋于稳定，也就是技能上的第一次瓶颈，这种人干活熟练，切页面可能也很快，代码看上去也比较规范，属于熟练工，开 始注重沟通技巧和一些职业技能的积累，比如带人带项目，至少有这方面的意识，并有过推动项目、和业务方pk需求的经历，这就达到了中级应当具备的职业技 能，但应当注意的是，这时最容易出现偏科的情况，特别是对于那些“专门切页面的“和“专门写脚本的“人，毕竟html/css/js三者不分彼此，三者是 一个合格前端工程师都必须要掌握的。如果你觉察到自身有偏废的嫌疑，则要小心了，要清楚的了解自身的差距，并意识到瓶颈的存在，为过渡到“中级“的打下基 础。

过了这道坎之后，工作3年以上的人大部分技能也趋稳，有些人对前端新技术有钻研，能够熟练应对日常工作，软技能也ok，具备有针对性的“拿来主 义“，代码也具有一定的架构性，开始突破“代码民工”的这一层瓶颈，对团队气氛、培训、工作环境有个性化的要求，一般来讲，这种人是典型的具有潜力的“中 级”工程师，但很快会遇到职业发展中的第二个技术瓶颈。

有少数工作3年或4年以上，在不断寻求新的技能上的突破，最明显的一点体现是，开始关注“底层协议”，即HTTP、第三方应用、系统对接、制造 工具、工作流程等，这时思考的重点已经脱离了“切页面”，变为“出方案“，比如要架设一个站点，能够搭建站点框架，预见站点后续（前端）开发中的所有风 险，并一一给出解决方案。项目后续开发遇到问题只要翻阅你提供的“手册”即能找到答案。这种人是标准的“高级”Web前端工程师。

出方案是一件挺难的事情，它要求一个工程师同时具备经验、技术、气场等诸多硬技能。尤其是对技术底子的要求非常高。

【半路出家】

那么，转行做前端的人又当如何呢？其实发展轨迹和科班秀才们非常类似，只是时间跨度可能会长一些，你要花更多的精力、做更多的项目、更多的反思和总结才能理解某个知识点的本质（比如HTTP协议）。当然这只是一般情况。

此外，这些人还需要摆脱很多思维定势的禁锢。这里我推荐大家阅读阿当的《Web前端开发修炼之道》。当然，如果你有一个靠谱的师兄带你入道，自然幸运万倍。

但不管怎样，我始终认为应当秉承兴趣第一的原则，不管你是误打误撞、还是意欲为之，不管你是科班秀才、还是半路出家，兴趣始终应当是第一原则， 然后才是你“想做好“。我对自己的要求无法强加于人，所以很多业界大牛在回顾自己成功之路时，提到最多的是：“热爱你的工作、拥抱它给你带来的挑战”。 N.C.Zakas曾经这样勉励大家：

“我对Web开发人员最大的建议就是：热爱你的工作。热爱跨浏览器开发带来的挑战、热爱互联网技术的种种异端，热爱业内的同行，热爱你的工 具。互联网发展太快了，如果你不热爱它的话，不可能跟上它的步伐。这意味着你必须多阅读，多动手，保证自己的才能与日俱增。下了班也不能闲着，要做一些对 自己有用的 事儿。可以参与一些开源软件的开发，读读好书，看看牛人的博客。经常参加一些会议，看看别人都在干什么。要想让自己快速成长，有很多事儿可以去做，而且付 出一定会有回报。“

第三日，幸福感

【先精通十行？！】

兴趣第一，听上去很美，但现实却不总是这么酷。练就了一身本领，那也要找到对口的怪物来打一打才过瘾。

自然，每个人都想做出好东西，每个工程师也都渴求这样的机遇，用层次分明的设计、漂亮优雅的代码、精妙的细节雕琢，做出美观、安全、实用耐用的 产品，不过现实是如此残酷，以至于工程师们一直都缺乏对产品的归属感。作为前端工程师，如何才能在江湖中把握住前进方向、步步走高？毕竟，在职位繁杂的大 公司，缺乏人性化的工作流程影响着工程师的工作幸福感。产品从设计之初、到技术方案评审、再到实现，处处充满了妥协，大部分产品都是杂交的产物，人与人相 互掣肘，每个人都对产品不满意……，大跃进式的敏捷开发早就被证明百害无一利。但，或许这就是成长的代价。年轻的工程师需要更多的了解需求和设计、产品经 理更要懂得软件迭代规律。对于前端工程师来讲更是如此，多学习交互设计和UI，多了解网络协议和软件迭代模型，更能帮助前端工程师和需求方沟通、和后台的衔接、以及控制版本的迭代。

说来奇怪，前端工程师不是写html/css/js的吗，搞懂那些边缘知识有什么用？《Web前端开发修炼之道》中也提到，精通一行需要先精通十行。这里我来解释一下原因。

作为交互设计师的下游，前端工程师学需要习设计知识是很容易理解的，因为它能帮助你更准确的理解设计师的意图，在原型不完整的时候也能正确的反 馈设计缺陷，将问题阻挡在设计的环节，会大大减少UI bug数量，比如说，设计师会给出理想状态下的容器样式，却往往忽略了文字溢出折行、长连续字符、容器宽高是否适应内容尺寸变化而变化，溢出部分是作截字 还是隐藏等诸多细节，因为设计师不懂“边界值测试”的道理，而这些问题往往在测试阶段才被发现，所以，如果能在拿到UI设计稿时就提醒设计师补充完整这些 场景，自然减少测试回归次数。

另外，前端工程师必须要了解网络协议，原因很简单，我们做的产品运行在Web上。很多依赖于Ajax的实现，只有前端工程师才会提出实现方案， 产品经理不了解技术瓶颈，后台工程师更不会在意客户端的用户体验，举个简单的例子：通过JS实现一个Ajax，如果Ajax抓取的数据源是一个302跳 转，则需要在JS程序中多做一些事情，这就需要前端工程师了解一些HTTP协议。应当说，这是很常见的一个场景。

那么，为什么说前端工程师也要关注代码版本控制呢？因为web开发和软件开发本质无异，同样具有迭代周期，需求不是一揽子提完、一口气开发完 的，是有步骤的开发，因此，每次上线开发哪些功能、为后续扩展功能留足哪些接口、代码在可扩展和可维护性上应当作哪些考虑……，这些应当是每个工程师关注 的事情，所谓迭代就是指这种需求的叠加，这是软件开发的常态，也是web开发的常态，刚开始，前端工程师总会不断抱怨没完没了的需求，代码起初还算干净， 但很快就越来越乱，代码的版本管理对于Web前端工程师来说有些困难，这也使得大部分前端工程师很难上档次，从这个角度讲，前端工程师是需要向后台工程师 学习的，他们的开发量不比前端少，维护代码的能力要超过前端工程师。另外，对于刚入行的前端工程师，心态要放对，提需求是产品经理的职责所在，整理出有价 值的需求是交互设计师的职责所在，将需求作版本控制分步实现是前端工程师的职责所在，前端工程师没必要去抱怨产品经理提一大堆没规律的需求，而更应当去理 解需求缘由，将需求提炼成UC（用例），让需求在自己手中可控制。只是多数前端工程师缺乏提炼、整理需求的能力，一味的在接需求，才会搞的手忙脚乱，带着 情绪堆代码。

所以，只有练就了一身本领，才会更有目标的去寻找对产品的责任感和对团队的归属感，不要误以为能切出漂亮的页面就是能力的提高，纯粹

的写代码每 个人都差不多的，要成为合格的工程师，眼界要进一步放开，前端工程师能做的，不仅仅是切页面而已，作一个精品项目，一定不乏专业的过程把控，这也是大多数 人最易忽略的地方。

【励志之本】

其实，除了个人需要明确努力的方向，每个人都更渴望身处一个好团队，谁都不希望有猪一样的队友。我们都很羡慕处身这样的团队，可以放心的将精力 放在纯粹的技术上，身边每个人都自觉的补充文档注释，代码也层次清晰解偶充分重用率高，精妙的设计实现可以更快的传播，bug得到的改进建议也是务实专业 的，技术在这种良性互动中价值倍增。我想这也算是好团队的一种境界了，这有赖于团队成员水平水涨船高。不过，反观Yahoo的成长之路，他们的技术积淀也 是靠点滴的积累，其实他们当初的状况不比现在的我们好哪去，10年的进化，才造就了Yahoo技术团队的专业性和Hack精神，我们每个人才刚刚起步而 已。为了积攒工作中的幸福感，多付出一些是值得的。

但我猜，你现在的处境一定不会太过乐观，产品乱提需求、一句话的PRD、不被重视，被生硬的当作“资源“……反正，情况就是这么个情况，要么你 选择抱怨下去，要么想办法去改变。“积极主动“是源自内心的一种坚韧品质，也是励志之本，有些人在现实中被磨平了理想，有些人却在黑暗森林中找到了方向， 这就是犬儒主义和英雄气概之间的差别。这自不必详说，因为这让我想起了“大长今”，这简直就是前端工程师的励志典范：“这是一个可怕的环境，足以消磨任何 人的斗志和信念，所有来这里的人都变得麻木和无所作为，‘多栽轩‘恶劣的环境没有改变长今，但长今却改变了‘多栽轩‘所有的人“。

如果你想做到“资深”，就一定要想清楚这一点，因为你是团队的顶梁柱（业务），也是幸福感的源头（士气）。

第四日，架构和伪架构

【代码设计的本质】

读到这里，你不禁会问，前端领域存在“架构师”吗？这个问题会在后面的“码农的宿命”中展开解释。这里先说下代码架构的一些琐事吧。

什么是架构？架构是由“架”和“构”组成，架，即元件，构，即连接件。因此，架构即是将总体分解为单元，然后定义单元之间的连接方式。架构的含 义源自禅宗，而禅宗的基本信条则之一就是真理是无法用语言来描述的。这个基本信条有其背景，即语言具有某种抽象性。而人们对这种抽象性的悟道则直接影响对 事物的看法，进而决定了对客观世界的分解方法。

而在编程语言中，同样存在这种禅宗所隐喻的悖论。在面向对象的教科书中，通常举一些显而易见的例子，比如“水果”是一个类，包含有苹果、桔子、 香蕉等实例，“蔬菜”也是一个类，包含白菜、冬瓜、茄子等实例。这两个类之间并无交集，因此很容易理解。但实际项目中情况要复杂的多，比如两个图书类目 “文学”和“历史”，那么“明朝那些事”应当是“文学”类的实例还是“历史”类的实例呢？即一旦用语言说出了某一事物，即人为的割裂了世界，于是就会陷入 迷途。这在程序设计领域情况更甚，也是造成混乱的主要根源，也就是说，如果你的程序可扩展性不好，一定是程序作者对“单元”的定义不够准确，即单元的概念 之间不够“正交”。而这种架构终是徒有其形，根基不稳。

因此，变量和类的命名才是真正考验架构功力的关键（命名是否准确清晰、单元之间是否有概念重叠或盲区），而和所谓“组合”、“继承”、“桥接”等模式化的“外表”无本质联系。

【伪架构】

实际情况是，程序员早早的就想让自己和“架构”扯上关系，并自封xx架构师。在项目中应用各种模式分层、解耦方法，每个项目都可以产出一套看上 去很复杂的“架构图”，感觉很牛逼的样子，没错，实践这些方法论总不是坏事，但世界观才是方法论的基础，只有在概念上对产品模块有科学的定义，方法论便自 然形成了，《编程珠玑》中一再提及数据结构就是静态的算法，在Web前端领域亦是如此，在页面的建模过程中，定义分解维度要比分解方法更加基础和重要。我 想阿当可以在《Web前端开发修炼之道》的第二版里加上这部分内容。

真正的高手用记事本就能写出高质量的代码、用cvs就能做到完美的版本控制、用字典式的分解就能做好系统架构，我想，这正是剑宗一派的最高境界吧。

第五日：寻找突破

【动心忍性】

技术流派看上去是如此吸引人，高手就像侠客一般，来去如风潇洒自如。但反观自己怎么看怎么没有侠客那股范儿。尽管上文提到了一些道理，了解这些 尽管不是坏事，但缺少实践总感觉是纸上谈兵。更何况，日常的工作又是枯燥无味、繁杂单调。每个人都盼望更高的目标、接触新鲜技术、将新技术运用到日常，在 探索尝试之中寻找成就感。这种感觉可以理解，但却缺少更深层次的思考。因为越到最后越会发现一线的工作才是最有挑战的。当然，我说这话的前提是，你能如前 文所说具备合格的软技能，需要一些技巧让工作变得工整有序、节奏健康，这样你才能将注意力放在纯粹的代码中，摆脱了外界的烦扰，方能从技术的角度思考突 破。这也是从初级到高级的进化过程需要大量的历练的原因。正如玉伯所说，“枯燥是创新的源泉。如果你发现自己没什么新想法，做事缺少激情，很可能是因为你还未曾体验过真正的枯燥的工作”。

关于如何寻找突破，我的建议是马上动手做、不要等，相信自己的直觉（这里和上文提到的先思后行是两码事）。比如，Slide幻灯控件理应支持触 屏事件以更好的适应移动终端，或许你在用的Slide幻灯版本很旧、或者时间不允许、再或者你害怕对Slide改造而引入bug，不要担心，大不了多花业 余时间，只要想，只要感觉合理和必要，就去做。因为这个过程带来的编程体验才是工程师们独有的美妙体味。我现在还时常深夜写代码，没有打扰、思如泉涌、代 码也更加工整严谨，不失为一种享受。因此，用眼睛去观察，用心去感触，“所以动心忍性，才会增益其所不能”啊。

【得与失】

互联网的发展的确太快，Web前端技术也在花样翻新，有人经不起诱惑，开始做新的尝试。前端技术虽然范围广，但各个分支都还比较容易入门，比如 服务器端脚本编程、再比如纯粹的WebApp，我认为这两者都是前端技术的范畴，毕竟他们都没有脱离“浏览器”，或者说类似浏览器的环境。NodeJS依 赖于V8，WebApp更是软件化的WebPage。只要打好基础，这些方向都是值得深入钻研的，因为，互联网的形态越发多元，新的技术总能找到用武之 地，这就要凭借自己的技术嗅觉和产品直觉，寻找技术和业务的契合点。

这看上去是一种放弃，放弃了自己赖以生存的铁饭碗（熟练的切页面至少不会失业），实则不然。这种想法是一种误区，新的选择并不会让你放弃什么， 就像学会了开车，并不意味着就不会骑车了。其实改变的是思维方式而已，是一种进步，如果你能想通这一点，你也能跟得上互联网发展的脚步了，打开你的思维， 让技术变成你的金刚钻，而不是包袱。

所以，所谓得失之间的权衡，其实就是“解放思想”。做到了这一点，那么你已经在做“技术驱动”了。

【误区】

但是，不要高兴的太早，“技术驱动”是需要大量的积累和经验的。在入行初期，很多人过于着迷与此，从而陷入了迷途。比如有人纠结于是否将dt、 dd的样式清除从reset.css中拿掉，原因是觉得这两个标签的清除样式会耗费一些渲染性能；或者是否需要将for循环改为while循环以提高js 执行速度。尽管这些考虑看上去是合理的，但并不是性能的瓶颈所在，也就是说，你花了很大力气重构的代码带来的页面性能提升，往往还不如将两个css文件合 成一个带来的提升明显。就好比用一把米尺量东西，没必要精确到小数点后10位，因为精确到小数点后2位就已经是不准确的了。这种技术误区常常让人捡了芝麻 丢了西瓜。

话说回来，这里提到的怀疑权威的精神是绝对应当鼓励的，但不应当止于表象，如果怀疑dt的清除样式会对性能带来影响，就应当想办法拿到数据，用事实来证明自己的猜测。数据是不会骗人的。而求证过程本身就是一种能力的锻炼。

【技术驱动】

说到这里，你大概对“技术驱动”有那么一点点感觉了。身边太多人在抱怨“公司不重视前端”、公司不是技术驱动的、技术没机会推动产品业绩、我的价值得不到体现?

什么是技术驱动？简单讲，就是技术对业务有积极推动作用。更多的是工程师发起、工程师影响、工程师负责。刚才提到的用数据说话只是一种“驱动”技巧，那么我需要何种数据，数据从哪里来？我来分享一个实际的场景吧。

工程师A被委派一个重要的频道首页，因为是新年版，所以要赶在年前上线。A学了一点点响应式设计，想在这次重构中加上，但谁也没做过响应式设 计，需求方根本不懂，设计师也懵懵懂懂，交互设计师太忙，做完交互稿就忙别的去了。A纠结了，按部就班的把项目做完上线发布，尽管不会出什么问题，但总觉 少点什么。这时A做了两个决定，1，我要按时完成项目，2，趁机实践我在响应式设计中的想法和思考，若成功，作为附加值赠送给需求方，若失败，权当技术玩 具耍一耍罢了。所以A熟练的提前完成了项目，剩下的时间开始考虑如何将首页适应到各个平台中，视觉设计是一大难题，他用吃饭的时间找了设计师收集建议，对 窄屏中的内容模块做了看似合理的编排，代码上hack一下，能够正确适配，就发布上线了。这件事情需求方不知道，视觉设计师也不了解，交互设计师更没工夫 操心。A感觉挺爽，开始给工程师弟兄们到处炫耀这个好玩的功能，B看了问，手机端访问量如何，A觉得这个问题有道理，就去部署埋点，一周后拿到数据出奇的 意外，首先，移动段的访问量稳步增加，趋势健康，再者，移动端首屏焦点广告位的点击率较PC端高了近一倍，这个数据让A喜出望外，兴奋的拿着报表找到交互 设计师C和市场研究的同事D，D看了报表之后立即启动一个项目，专门调研公司全站响应式设计页面在PC端和移动端的点击率、PV、UV趋势方面的影响…… 后来发生的事情就都水到渠成了，设计师C开始注意设计页面交互时（至少是有条件的考虑）对移动端的适配，D的调研报告也放到了UED老大的案头……接下来 的事情，你懂得。A被指派要出一套响应式最佳实践和规范，最终，A走在了技术的前沿，也因此拿到了好绩效。

这件事情就是一个典型的技术驱动的例子。谁不让你玩技术了，谁不重视你了，谁把你当工具了，谁觉得你的代码没价值？这世界只有自己把自己看扁，谁想跟你这个蝇头小卒过不去？用实力说话，用数据说话，用独到的见解说话，想不做技术驱动都难。

第六日：码农的宿命

【青春饭】

“码农”是IT从业者一个自嘲的称号，也有从事没有发展前景的软件开发职位，靠写代码为生的意思。但我认为码农是一个爱称，编码的农民，和农民 一样有着执着纯真朴实豪爽的共性，仅仅分工不同而已。就好比农业社会对粮食的依赖，工业化进程对计算机应用也有着很强的依赖，大量的需

求催生出这样一群 人。他们有智慧的大脑，对于编程，设计，开发都有着熟练的技巧，但多数人看来，码农的特点是：

1，收入低
2，工作单调
3，工作时间长

实际上这个描述非常片面，或者说是外行看热闹。第一，全行业比较来看，软件开发领域收入为中等偏上；第二，程序员一般都是有癖好的，沉浸在自己 的癖好中是不会感觉单调的；第三，程序员有一定的时间自由度（如果你是一名合格的程序员的话），至少不会像流水生产线工人一样。其实，通过几十年的发展， 我们对程序员的定义更加科学，比如很多IT企业都开始建立详细的JM（Job Module），即职级模型，程序员沿着专业方向可以走到很高，甚至可以说，程序员是可以被当成一生的事业的。

然而，有一个非常普遍的观点是，程序员和做模特一样是吃青春饭的，到了三十岁就要考虑转行或者转管理。尽管这种观点颇具欺骗性，但至少它对一种 人是适用的，即入错了行的人。如果你骨子里不想写程序，就算年纪轻轻为了生计写几年代码，之后肯定会另有他途。心非所属则不必勉强，但问题是，即便如此， 你知道你的心之所属吗?

我们知道，一个成熟的产业一定需要各色岗位来支撑，若要成熟，则需要时间的沉淀，比如实体经济制造业，创意、生产线、高级技工、技术管理四个方 面都产出大量的高级人才。因为历史悠久，我们能看得到。而软件产业则不然，九成以上是刚出道的新手，并没有太多“高级”和“资深”的具体样板可供参照，在 前端开发领域中情况更甚，绝大部分人根本搞不清楚什么样才是“资深”前端工程师，相比传统软件行业近四十年的进化，我不相信仅有几年光景的前端技术岗位能 产出多少货真价实的“资深”。但互联网崛起速度太快，还没有等技术基础打牢，互联网形态就又花样翻新了，这种变化是一种常态，而岗位的设定也在这种变化之 中自然的优胜劣汰，比如两年前可能还难以想象数据部门会需要前端工程师，他们甚至不直接和浏览器打交道。前端工程师需要适应这种变化带来的观念冲击，不要 以为自己只能做切页面、或者只会给页面搞重构、只会搞兼容性，要把自己放在整个软件行业来看。

所以，由于历史“不悠久”导致的岗位模糊本身不是什么大问题，岗位的演化本身就包含在互联网的发展轨迹之中。所以，当今的互联网IT状况，就好 比移动终端的大哥大时代、云计算的肉鸡时代、或者桌面操作系统的DOS时代。因此，前端工程师当前要务是要想清楚看清楚，在互联网中我能做什么，而不是作 为前端工程师我能做什么，所以，从这个角度讲，技术是一个工具，放大来看，技术也只是你职业生涯中很小的组成部分，而你的从业积累、和知识面的广度深度才 是你随着时间的推移慢慢步入“资深”的原因所在，而不是写了个什么框架就变“资深”了。如果有一天互联网形态固定了，它的岗位可能真正就定型了，才会有真 正清晰的职能边界，就像蓝色巨人IBM中的各色岗位一样，边界清晰，权责分明，普通程序员只能实现接口而无机会设计接口、低层级的工程师也无机会跃进式的 接触项目架构、技术经理人也不能轻易对产品有决策性影响，到这时，人的能力才真正的被限制在方圆之内，容不得越界，这种环境下人的成长非常缓慢。根本不会 有像今天互联网乱局之中所提倡的创新、革命、成长和思想解放。简单讲，一旦产业定型，就不太需要很多“创造”了，更多的是“维护”。所以，我个人宁愿互联 网IT“黑暗”的中世纪越久越好，至少对于年轻气盛程序员来说，黑暗的丛林环境才是真正的自然进化最理想的土壤，这时我想起了狄更斯在“双城记”中的开 篇。

“这是最好的时代，这是最坏的时代；这是智慧的时代，这是愚蠢的时代；这是信仰的时期，这是怀疑的时期；这是光明的季节，这是黑暗的季节；这是希望之春，这是失望之冬；人们面前有着各样事物，人们面前一无所有；人们正在直登天堂，人们正在直下地狱”。

【半路出家的危与机】

然而，不管怎样，信心的树立不是一蹴而就的，对于转行做前端的人来说更是如此。俗话说，隔行入隔山。每个行业自有其道，自然不是想做就做。前端 技术领域半路出家者非常多，我们来分析一下转行的心理。第一，看到前端技术入门简单、互联网对前端技术的需求缺口巨大；第二，前端技术所见即所得、感觉学 习起来很快；第三，我身边的某某转行作前端看上去不错、我似乎也可以；第四，我不喜欢我现在做的工作、想换行业、正好前端技术上手较快，就选他吧；第五， 我真的喜欢做Web前端，为它付出再多都是值得的。

转行者的心态比较容易走两个极端，一是只看到新行业的好，二是只觉得原工作很糟糕。但不管是什么行业的转行，对自己的职业规划的思考都应当先行一步。即务必首先清晰的回答这些问题：

1，我能做什么？
2，我不能做什么？
3，我的优势是什么？
4，我的劣势是什么？
5，做新行业对我有何好处？
6，换行会让我付出何种代价？
7，如何定义转行成功？

因为面试的时候一定会被这些问题所挑战。如果支支吾吾说不清楚，要么是对自己未来不负责任，要么骨子里就是草根一族，习惯做什么都蜻蜓点水浅尝 辄止，也难让人信服你的转行是一个权衡再三看起来合理的选择。我无法帮每个人回答这些问题，但至少有两点是确定的，第一，Web前端技术是一个朝阳行业， 绝对值得义无反顾的坚持下去；第二，你将经历从未有过的枯燥、苛刻的历练，所谓痛苦的“行弗乱其所为“阶段。不过话说回来，经历过高考的人，还怕个屁啊。

有心之人自有城府、懂得放弃，看得清大势中的危机、识得懂繁华里的机遇。尤其当立足于Web前端技术时，这种感觉就愈发强烈。因为国内外前端技 术领域从2000年至今一直非常活跃，前端技术前进的步伐也很快，对于一些人来说，不管你是在大公司供职还是创业，不管你是在接外包项目还是自己写开源项 目，从转行到跟得上新技术的脚步是有一些方法和“捷径”的。

第一，梳理知识架构

我们知道知识积累有两种思路，第一种是先构建知识面、建立技术体系的大局观，即构建树干，然后分别深入每一个知识点，即构建枝叶，最终形成大 树。第二种是先收集知识点，越多越好，最后用一根线索将这些知识点串接起来，同样形成大树。第一种方法比较适合科班秀才，第二种方法则更适合转行作前端的 人，即实践先行，理论升华在后。比如对“IE6怪异模式“这条线索来说，要首先将遇到的IE6下的样式bug收集起来，每个

bug都力争写一个简单的 demo复现之，等到你收集到第100个bug的时候，再笨的人都能看出一些规律，这时就会自然的理解IE的hasLayout、BFC和各种bug的原 因、你就成为了IE6的hack专家了，当你成为100个知识线索的专家的时候，你已经可以称得上“资深”的水平了。我们知道，10个人中有9个是坚持不 下来的，他们会以项目忙等各种理由万般推托，将自己硬生生的限制在草根一族，坐等被淘汰。所以，对于立志作前端的人来说，这种点滴积累和梳理知识非常重 要。

第二，分解目标

将手头的工作分解为几部分来看待，1，基本技能，2，项目经验，3，沟通能力，4，主动性和影响力。想清楚做一件事情你想在哪方面得到历练，比 如，我之前在做第一次淘宝彩票常规性重构的时候（正好是一次视觉和交互上的全新设计），我清楚的明白这次重构的目的是锻炼自己在架构准富应用时的模块解偶 能力，寻找在其他项目中架构的共通之处，所以我宁愿加班或花更多精力做这个事情，当然更没打算向业务方多解释什么，这件事情对我来说纯粹是技能的锻炼。而 经过这一次重构之后，我意外的发现对业务的理解更透彻深入、更清晰的把握用户体验上的瓶颈所在。如果一开始就把这次常规改版当成一个普通的项目按部就班的 做，我只能说，你也能按时完成项目，按时发布，但真真浪费了一次宝贵的锻炼机会，项目总结时也难有“动心忍性”的体会。

所以，每个项目的每个事情都应当认真对待，甚至要超出认真的对待，想清楚做好每件事对于自己哪方面有所提升？哪怕是一个bug的解决，即便不是 自己的问题也不要草草踢出去了事，而是分析出问题原因，给出方案，有目的involve各方知晓……，正规的对待每个不起眼的小事，时间久了历练了心智， 这时如果突然遇到一个p0级的严重线上bug（比如淘宝首页白屏，够严重的了吧）也不会立即乱了方寸，这也是我上文提到的心有城府自然淡定万倍，而这种淡 定的气场对身边浮躁的人来说也是一种震慑和疗伤，影响力自然而然就形成了。

第三，作分享

做分享这事儿真的是一本万利。有心的人一定要逼着自己做分享，而且要做好。首先，自己了解的知识不叫掌握，只有理解并表达出来能让别人理解才叫 掌握，比如如果你解释不清楚hasLayout，多半说明自己没理

解，如果你搞不懂双飞翼的使用场景，可能真的不知道布局的核心要素。再者，作分享绝对锻 炼知识点的提炼能力和表达能力，我们作为工程师不知道多少次和强硬的需求方pk，被击败的一塌糊涂。也反映出工程师很难提炼出通俗易懂的语言将技术要点表 述清楚。而做ppt和分享正是锻炼这种能力，将自己的观点提炼出要点和线索，分享次数多了，自然熟能生巧。档次也再慢慢提高。另一方面，逼迫自己站在公众 场合里大声讲话，本来就是提高自信的一种锻炼。

这时，你或许会问，我讲的东西大家都明白，我讲的是不是多余，我第一次讲讲不好怎么办，大家会不会像看玩猴似的看我“这SB，讲这么烂还上来讲”？要是讲不好我以后再讲没人听怎么办，我今后怎么做人啊？

老实说，这是一道坎，任何人都要跨过去的，谁都一样，你敢鼓起勇气在大庭广众之下向爱人表白，就没勇气对自己的职业宿命说不？其实勇敢的跨越这 一步，你会意外的收获他人的掌声和赞许，这些掌声和赞许不是送给你所分享的内容，而是送给你的认真和勇气。这个心结过不去，那就老老实实呆在自己的象牙塔 里遗老终生，当一辈子工程师里的钻石王老五吧。

【匠人多福】

如果你能耐心读到这里，心里一定有一个疑问，上面说的都是技术上能力上怎样怎样，那我所做项目不给力又当如何？如果项目不挣钱、黄了、裁了，我的努力不就白费了吗？我又有什么绩效和价值呢？

没错，有这种想法的人不在少数。特别是刚出道的校招同学往往更加心高气傲，以为自己有改变世界的本事，一定要参与一个牛逼的团队做一款光鲜靓丽受人追捧能给自己脸上贴金的项目。如果你有这种想法，趁早打消掉这个念头，当然，我们这里先不讨论创业的情形。

第一，如果你刚毕业就加入一个牛逼团队，说难听点，你就是团队中其他人眼中的“猪一样的队友”，不创造价值且拖项目后腿（显然大家都要照顾你的成长啊），按照271理论，你没有理由不是这个1。至少相当长一段时间内是这样。

第二，你在所谓牛逼团队中的创造性受限，因为创新多来自于团队中的“资深“和大牛们，你参与讨论但观点通常不会被采纳，他们只会给你这

个菜鸟分活干，想想看，你如何能花两到三年就超越身边的大牛们？甚至连拉近与他们的距离都难。

第三，如果身在牛逼团队，自然心理对周围的牛人们有所期待，希望他们能灌输给你一些牛逼的知识和牛逼的理念。这种思想上的惰性在职场生涯之初是非常危险的。要知道技术和知识本身是很简单和淳朴的，只不过披上了一个光鲜项目的外衣而让人感觉与众不同。

第四，由简入奢易，由奢入简难，做过一个看似光彩的项目，心理再难放平稳，去踏实的做一个看上去不那么酷的产品。这种浮躁心态会严重影响今后的职业发展和成长。

第五，光鲜靓丽的项目被各种老大关注，是难容忍犯错误的，傻瓜都知道犯错误在成长之初的重要性。

就我所看到的情形看，一开始加入看似很牛的项目组，三年后得到的成长，比那些开始加入一个不被重视的项目的同学要小很多，而后者在能力上的弹性 却更大。所以，道理很简单，你是要把一个很酷的项目做的和之前差不多酷，还是把一个不酷的项目做的很酷？项目是不是因为你的加入而变得与众不同了？

从这个角度讲，不管是转行的新人还是刚出道的秀才，最好将自己当作“匠人”来对待，你的工作是“打磨”你的项目，并在这个过程中收获经验和成 长。付出的是勤奋，锻炼的是手艺，磨练的是心智。因此，你的价值来自于你“活儿“的质量，“活儿”的质量来自于你接手的项目之前和之后的差别。做好活儿是 匠人应有的职业心态。想通这一点，内心自然少一些纠结，才会对自己对项目的贡献度有客观的认识，不会感觉被项目所绑架。

做一名多福的匠人，拥有了金刚钻、就不怕揽不到瓷器活儿。但对于人的成长来说，如果说“项目”重要但不关键，那么什么才是关键呢？这个话题还会在接下来的“伯乐与千里马”这篇中给出答案。

【若干年后】

现在，让我们回过头回答一下“青春饭”的问题。在“青春饭”小节中提到，“程序员到三十岁之后需要转行或者转管理吗？”

上文提到，工业化生产的四个领域，1，创意，2，生产线，3，高级技工，4，技术管理。Web前端技术也是如此，可以在这四个领域找到各自的归宿。

第一，“创意“

即和产品需求越走越近，拥有良好的产品感，对产品需求、设计交互把握准确，能够用适当的技术方案推动产品用户体验，属于“架构师”的范畴，因为 职能更加靠前，偏“出主意”型的。这种人更贴近用户，需要活跃的思维、广阔眼界、厚实的项目经验。更多的影响产品体验方面的决策。

第二，“生产线“

即前端基础设施建设，优化前端开发流程，开发工具，包括开发环境、打包上线自动化、和各种监控平台和数据收集等，属于“技术支持”的范畴，相比于很多企业粗犷难用的平台工具，前端技术方面的基础设施建设基础还需更加夯实，因为这是高效生产的基本保证。

第三，“高级技工“

即高级前端开发工程师，专职做项目，将产品做精做透，用代码将产品用户体验推向极致，偏“实战”型的，是项目的中坚力量，直接产出成果，影响产品效益。属于项目里的“资深”。

第四，“技术管理“

即做技术经理，这才是多数人所理解的“管理”，其实就是带团队、靠团队拿成果。这类人具有敏感的技术情结，在技术风潮中把握方向，能够指导培训新人，为各个业务输出前端人才，偏“教练”型的，促进新技术对业务的影响。并有意识的开辟新的技术领域。

可见，转管理可不是想当然，也不是所谓做项目变资深了就能转管理，转了也不一定能做好。根据“彼得原理”，即人总是倾向于晋升到他所不能胜任的岗位，这时就又陷入“帕金森”定律所隐喻的恶性循环之中，直到你带的团队整个垮掉。

所以，转管理应当是一件非常慎重的事情，不是所谓程序员混不下去就转管理这么简单。但不管怎样，有一件事情是需要尤其要想清楚，即，

转了管理，技术就丢了吗？我们在第七日“伯乐与千里马”中再深入聊聊这个事儿。

第七日，伯乐与千里马

【师兄们的抉择 1】

千里马常有，而伯乐不常有。——韩愈，“马说”。

一个人这辈子能遇到一个好师兄是一种缘分，可遇不可求。很多人工作中的幸福感似乎也源自这种被认同，被师兄的了解和认同，有人能直言不讳的指出 你的不足，帮你发现机会，并将最适合你做的事情分配给你，这是莫大的幸运，但如此幸运的人十之一二，大多数人因为缺少伯乐的提点，渐渐辱于“奴隶人之手 “，潜力渐失，毁于中庸。

在前端技术领域，这种情况很普遍也很特殊，当然有很多客观原因。即前端技术进入公众视野时间不长，有实力的伯乐更加是凤毛麟角。更何况，Web 前端技术还有着一些江湖气，知识点过于琐碎，技术价值观的博弈也难分伯仲，即全局的系统的知识结构并未成体系，这些因素也客观上影响了“正统“前端技术的 沉淀，奇技淫巧被滥用，前端技术知识的传承也过于泛泛，新人很难看清时局把握主次，加之业务上的压力，未免过早导致技术动作变形。而这些问题也无法全赖自 己全盘消化，若有人指点迷津，情况要好上万倍。因此，前端技术领域，为自己觅得一个靠谱的师兄，重要性要盖过项目、团队、公司、甚至薪水。

这也是上文所说的“项目不重要，师兄才重要“的原因。说到这里就有一个问题，每个人都问下自己，你是想当师弟呢还是想当师兄呢？当师兄有什么好处呢?

没错，很多师兄都是被师兄，甚至没有做好当师兄的准备，更进一步说，不少经理人也都是“被经理人“，没有做好准备就被推到了管理岗位。带人是耗 精力的，师兄要做很多思想斗争才舍得把这些宝贵的精力放在那些菜鸟身上，这不是一个技术问题，而是一个道德问题。要记住，没有谁应该无缘无故把自己所掌握 技能给你倾囊相授，如此皆命也。读到这里，作为菜鸟，作为学徒，作为新人，作为师弟，你做到对这份命运的足够尊重了吗?

尊师重教的传统美德并没有在技术领域得以很好的延续。也正因为此，人才梯队难建立起来，但对于师兄来说，却是有更多机遇的。

【师兄们的抉择 2】

作为师兄，不管是主动还是被动，肯定会想当师兄对我有什么提升？对于初次做师兄的人来说，最大的提升在于两方面，1，任务分解，2，问题分析。

第一，任务分解，作为师兄要给师弟派分任务，就涉及到任务分解，分解这事儿往低了说，就是派活，往高了说，其实就是做“架构”，比如一个页面， 按照什么思路进行模块划分，模块划分是否适合单人开发，如何控制共用样式和共用脚本，我需要为他提供什么接口，如何控制他的代码并入整个页面时不会影响整 体页面代码的熵值，这些都是实打实的“架构“应该包含的问题，而从小页面开始就做这种锻炼，做的多了，“架构感”自然就形成了。

第二，问题分析，在之前自己写代码都是单打独斗，什么都是用代码解决问题，但一旦涉及协作，就要强迫自己分析问题，或者说给徒弟分析问题，告诉 他应当用什么方法来解决问题，当说到“方法”时，脑子里定形成了一个方案，按照这个方案路子走一定能解决问题。分析问题比写代码要更抽象、更高效，因为在 脑子里构建方案要比写代码要快，思考也会更加缜密，当锻炼的多了，思考越来越快，代码的草稿也很快就在脑海中形成了，这也是我们说为什么很多人不写代码但 编码思路和水平都很高的原因。

这些工作方法对了，积累多了，就是提高。对于技术经理人来说，也是同样的道理。所以，就像在第五日的“得与失”部分提到的那样，转身师兄、变身管理并不意味着“失“掉技术饭碗，而是一种进步。

【做自己的伯乐】

那么，在前端技术领域里什么样的人才算千里马，其实人人都是千里马，人人都可以发掘自己的潜力，如果上面的文字你能读懂，能认可，这种自我发掘已经开始了，没有一个好伯乐又何妨呢？做一个勤快的小码农，少一些势利的纷争，很快会发现，自己才是最好的伯乐。

但这并不是说，他人对自己的观点不重要，有时甚至要综合各种声音，所以，多找身边的大牛们聊聊天，多找你的师兄和主管，不管他们给你的建议是多么形而上，总有一些声音对你是有益的，多收集，有好处。

第八日，做地球上最牛的UED

【谁推动了历史前进，英雄？还是人民？】

“做地球上最牛的UED！”，这是淘宝UED创立之初的口号，现在被渐渐淡忘了，因为微博上的一些讨论，又想起了这份曾经美好的初衷。玉伯也感 叹道：“这愿景曾吸引了多少好汉前往投奔呀。只可惜短短几年间，这愿景好像越来越远了”。问题是，要做好一个团队，靠的是个人、还是整体？愿景是越来越远 了吗？

是谁推动了历史的前进，是英雄？还是人民？微观来看，是英雄，宏观来看，是人民。再放大了看，是互联网大潮之崛起推动了前端技术的进步，时势需要UED、需要用户体验。

所以，UED团队的创立发展受这些积极的外因影响，赶上了好时候，成员也跟着沾光。然而，我并不关心这个口号，我只关心体制内的关键人物，那些 带动整个团队水涨船高的人们。往往我们发现，某些人的高度代表了整个团队的高度，个体的影响力代表了整个团队的影响力，某个人的水平代表了整个团队的水 平。支付宝、淘宝、腾讯、百度、盛大，都是如此。而我们作为普通的个体，正是要励志成为这种人，成为真真用技术推动用户体验前进的尖刀人物。

这时我想起了很多人在知乎上的问题，关于跳槽、关于转行、关于创业、关于各种UED团队。我想，读得懂我上面的文字，你心理也许会有自己的答案。

【归宿】

最后，还有一个不得不说的问题，即归属问题，前端开发应当归属于UED还是技术部门？应当说，当前Web前端技术的价值体现在“用户体验“上。 是用户体验这块阵地最后一道坎。也就是说，前端工程师应当重点考虑我所作的页面的感官体验。这是需要一些灵感和感性的，应当看到帅气优雅的界面会心有所 动、或者实现一款精巧的小组件时萌生一阵快意。这种所见即所得的美妙编程体验正是其他后端工程师无法体验到的。因此，这

种精确到像素级的精工雕琢虽然不直 接决定产品生死，但却是提升产品品味和时尚感的要素。物质越来越丰富的今天，大众的更高诉求不就是品味和时尚吗?

如果将前端归到技术部门，一方面和“设计“离的更远，代码写的规规矩矩但渐缺少了灵性，另一方面作为工程师又缺少计算机专业课的功底，才真正丧 失了优势所在，如果有一天，前端工程师的平均水平足够高，清一色的计算机科班出身，似乎更合适归入到技术部门。所以，Web前端工程师是“工程师“，需要 科学严谨的编程能力，但身处UED所应当具备的美感和灵性是万不可被剥夺去的。

还有一点，Web前端工程师作为UED之中最具实践精神和逻辑思维的工种，是能够将技术对设计的影响发挥到最大，可以催生出大量的创造和革新的，这一点也是传统后端工程师所不具备的。

第九日，前端技术体系

现在越来越感觉到前端技术需要成体系的积累，一方面可以规范我们的前端技术培训，另一方面，作为知识线索为新人做指引，省的走弯路，避免陷入奇技淫巧的深坑之中不能自拔。

之前我整理了一下“前端技术知识结构”，罗列的比较散，但也基本表述清楚了我的观点。今年上半年也在整个研发中心组织了一次前端技术培训，对于前端技术的演化规律也有过整理，都放在了这个ppt中，希望对大家有所帮助。

概观国内前端技术界，其实我不认为和国外顶尖的前端技术有多少年差别，甚至很多方面都走在了他们前面。

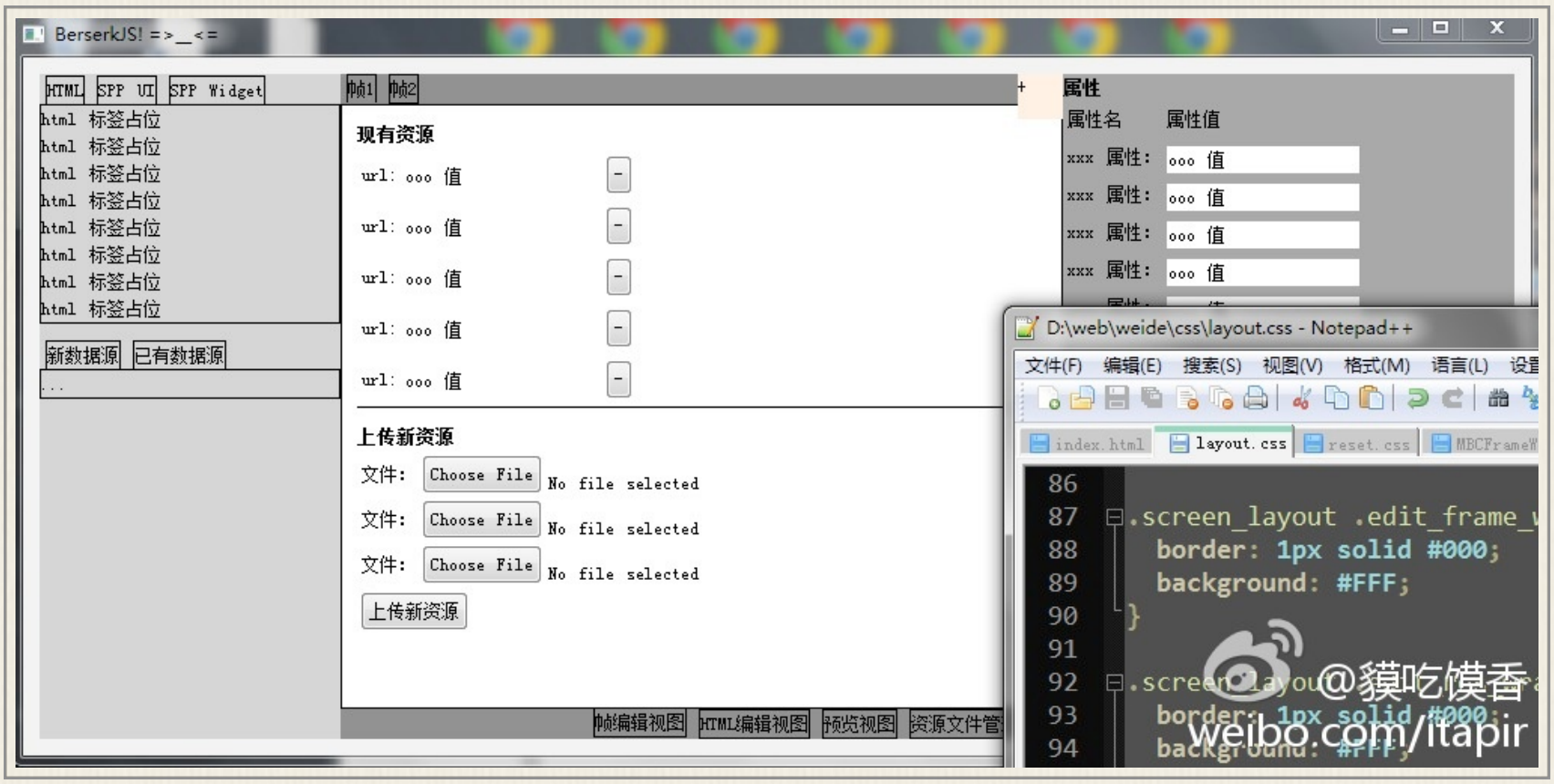
原文链接：http://jianshu.io/p/8cf2df3fdbf2

前端网络（性能）监测工具 berserkJS

作者：开源中国

berserkJS 是基于 Qt （C++跨平台库）开发的前端网络（性能）监测工具。 它的核心功能是通过内置 webkit 收集由页面实际网络请求相关数据。偏重于页面上线前检测与评估。

页面性能分析工具，可用 JS 编写自己的检测、分析规则。 基于 Qt 开发，可跨平台编译，部署。内置基于 QtWebkit 的浏览器环境。 源码需在目标系统中编译后，可产生运行于 Windows / Linux / Mac 系统的可执行文件。 工程中自带 Window 系统中动态编译的可执行文件，此文件位于 bulid 目录下。



使用案例

- 无界面浏览器测试：在不依赖本地任何浏览器的情况下，运行测试框架，如 QUnit，Capybara, QUnit, Mocha, WebDriver, YUI Test, BusterJS, FuncUnit, Robot Framework 等。

- 页面自动化：可以无障碍访问和操作网页的标准 DOM API 以及页面所用 JS 变量、对象、属性等内容。

- 屏幕捕获：以编程方式获取网页全部或部分内容，可根据 Selector 截取指定 DOM 元素渲染情况；包括 CSS，SVG 和 Canvas。可将截取图片 base64 化，以便发送给远端服务器保存。

- 网络监控：自动化的网络性能监控，跟踪页面所有资源加载情况并可简便的将输出结果格式化为标准HAR格式。

- 页面性能监控：自动化的页面渲染监控，可获取 CPU、 内存使用情况数据，根据页面整体情况可简便的输出首次渲染时间、首屏渲染时间等关键数据。

工具特性

- 跨平台性：基于 Qt 开发，可跨平台编译，部署。内置基于 QtWebkit 的浏览器环境。源码需在目标系统中编译后，可产生运行于 Windows / Linux / Mac 系统的可执行文件。

- 功能性：工具内置 webkit 浏览器内核，可响应浏览器内核事件回调、支持发送鼠标消息给浏览器、包装浏览器网络请求数据为JS数据格式、可与浏览器内JS做数据交互。

- 开放性：工具将主要操作均包装为JS语法与数据格式，采用JS语法包装，前端工程师可根据API组装出符合各自预期的检测功能。

- 接口性：工具本身支持命令行参数，可带参调用。API支持处理外部进程读取输出流、支持HTTP发送数据。可由 WEB 程序远程调用后获取测试的返回结果。

- 标准性：完全真实的浏览器环境内 DOM，CSS，JavaScript，Canvas，SVG 可供使用，绝无仿真模拟。

特点差异

与 PhantomJS 相比具有以下不同：

- API 简易: 更直接的 API，如获取网络性能数据，仅需 3 行代码，而非 PhantomJS 的几十行，且信息量比 PhantomJS 丰富。
- API 标准化： 常用 API 均采用 W3 规范标准命名，事件处理代码可重复绑定而不相互覆盖，可以无缝兼容 Wind.JS 等异步流程处理库来解决自动化时异步流程控制问题。
- 页面性能信息丰富：具有页面渲染和 CPU、 内存使用情况数据获取能力，可输出首次渲染时间、首屏渲染时间等页面性能关键数据。
- 调试便利: 具有 GUI 界面与命令行状态两种形式，开发调试期可使用 GUI 模式定位问题，此模式中可开启 WebKit 的 Inspector 工具辅助调试页面代码与 DOM 。实际运行时可开启命令行状态避免自动执行时 GUI 界面干扰。

应用企业

- 新浪微博：已使用 berserkJS 构建前端性能监测数据分析平台，防止微博主要产品在不停开发迭代时，页面性能产生退化。
- Cisco: 用于 WebEx 项目的自动化测试

原文链接：http://www.oschina.net/p/berserkjs

为什么要使用 Go 语言，Go 语言的优势在哪里?

作者：asta谢

1、Go有什么优势

- 可直接编译成机器码，不依赖其他库，glibc的版本有一定要求，部署就是扔一个文件上去就完成了。
- 静态类型语言，但是有动态语言的感觉，静态类型的语言就是可以在编译的时候检查出来隐藏的大多数问题，动态语言的感觉就是有很多的包可以使用，写起来的效率很高。
- 语言层面支持并发，这个就是Go最大的特色，天生的支持并发，我曾经说过一句话，天生的基因和整容是有区别的，大家一样美丽，但是你喜欢整容的还是天生基因的美丽呢？Go就是基因里面支持的并发，可以充分的利用多核，很容易的使用并发。
- 内置runtime，支持垃圾回收，这属于动态语言的特性之一吧，虽然目前来说GC不算完美，但是足以应付我们所能遇到的大多数情况，特别是Go1.1之后的GC。
- 简单易学，Go语言的作者都有C的基因，那么Go自然而然就有了C的基因，那么Go关键字是25个，但是表达能力很强大，几乎支持大多数你在其他语言见过的特性：继承、重载、对象等。
- 丰富的标准库，Go目前已经内置了大量的库，特别是网络库非常强大，我最爱的也是这部分。
- 内置强大的工具，Go语言里面内置了很多工具链，最好的应该是gofmt工具，自动化格式化代码，能够让团队review变得如此的简单，代码格式一模一样，想不一样都很困难。

- 跨平台编译，如果你写的Go代码不包含cgo，那么就可以做到window系统编译linux的应用，如何做到的呢？Go引用了plan9的代码，这就是不依赖系统的信息。

- 内嵌C支持，前面说了作者是C的作者，所以Go里面也可以直接包含c代码，利用现有的丰富的C库。

2、Go适合用来做什么

- 服务器编程，以前你如果使用C或者C++做的那些事情，用Go来做很合适，例如处理日志、数据打包、虚拟机处理、文件系统等。

- 分布式系统，数据库代理器等

- 网络编程，这一块目前应用最广，包括Web应用、API应用、下载应用、

- 内存数据库，前一段时间google开发的groupcache，couch-base的部分组建

- 云平台，目前国外很多云平台在采用Go开发，CloudFoundy的部分组建，前VMare的技术总监自己出来搞的apcera云平台。

3、Go成功的项目

nsq：bitly 开源的消息队列系统，性能非常高，目前他们每天处理数十亿条的消息

docker:基于lxc的一个虚拟打包工具，能够实现PAAS平台的组建。

packer:用来生成不同平台的镜像文件，例如VM、vbox、AWS等，作者是vagrant的作者

skynet：分布式调度框架

Doozer：分布式同步工具，类似ZooKeeper

Heka：mazila开源的日志处理系统

cbfs：couchbase开源的分布式文件系统

tsuru：开源的PAAS平台，和SAE实现的功能一模一样

groupcache：memcahe作者写的用于Google下载系统的缓存系统

god：类似redis的缓存系统，但是支持分布式和扩展性

gor：网络流量抓包和重放工具

以下是一些公司，只是一小部分：

- http://Apcera.com
- http://Stathat.com
- Juju at Canonical/Ubuntu, presentation
- http://Beachfront.iO at Beachfront Media
- CloudFlare
- Soundcloud
- Mozilla
- Disqus
- http://Bit.ly
- Heroku
- google
- youtube

下面列出来了一些使用的用户

https://code.google.com/p/go-wiki/wiki/GoUsers

4、Go还存在的缺点

以下缺点是我自己在项目开发中遇到的一些问题：

- Go的import包不支持版本，有时候升级容易导致项目不可运行，所以需要自己控制相应的版本信息

- Go的goroutine一旦启动之后，不同的goroutine之间切换不是受程序控制，runtime调度的时候，需要严谨的逻辑，不然goroutine休眠，过一段时间逻辑结束了，突然冒出来又执行了，会导致逻辑出错等情况。
- GC延迟有点大，我开发的日志系统伤过一次，同时并发很大的情况下，处理很大的日志，GC没有那么快，内存回收不给力，后来经过profile程序改进之后得到了改善。
- pkg下面的图片处理库很多bug，还是使用成熟产品好，调用这些成熟库imagemagick的接口比较靠谱

最后还是建议大家学习Go，这门语言真的值得大家好好学习，因为它可以做从底层到前端的任何工作。

学习Go的话欢迎大家通过我写的书来学习，我已经开源在github：

https://github.com/astaxie/build-web-application-with-golang

还有如果你用来做API开发或者网络开发，那么我做的开源框架beego也许适合你，可以适当的来学习一下：

https://github.com/astaxie/beego

原文链接：http://www.zhihu.com/question/21409296/answer/18184584

我终于深入参与了一个分布式系统了，好多想法不一样了！

作者：Turbo Zhang

前言

过去两个月深入的参与了一个分布式系统的开发，记得之前有人说过“想成为架构师之前，都是从微观架构开始的”。尽管我从没想过将来的某一天要成为一 个架构师，或者领域专家，我只是想萌萌哒的编码，写着自己喜欢的Code，和一群志同道合的朋友做出大家喜欢的商品和产品。但是工作久了慢慢的搭架子的事 情还是会来到你的面前，因为时间总会把一部分人慢慢推向海边，使得他们成为最早见到阳光的人。

不扯淡了，为什么要说阳光呢，还是因为过去的两（三）个月可能过的太充实也太痛苦了，完成之后，曙光来临的时候整个人是会发光的哦。“深度”参与是因为我终于有机会在搭架子的过程中有了话语权和选择权，同时也会承担70%以上的编码工作。

之前我的自我认知是我可能在软件方面的积累还可以，比如设计模式，架构分层，程序解耦，API入手等方面，但是总觉得我在硬件网络方面积累的太少，太薄了。

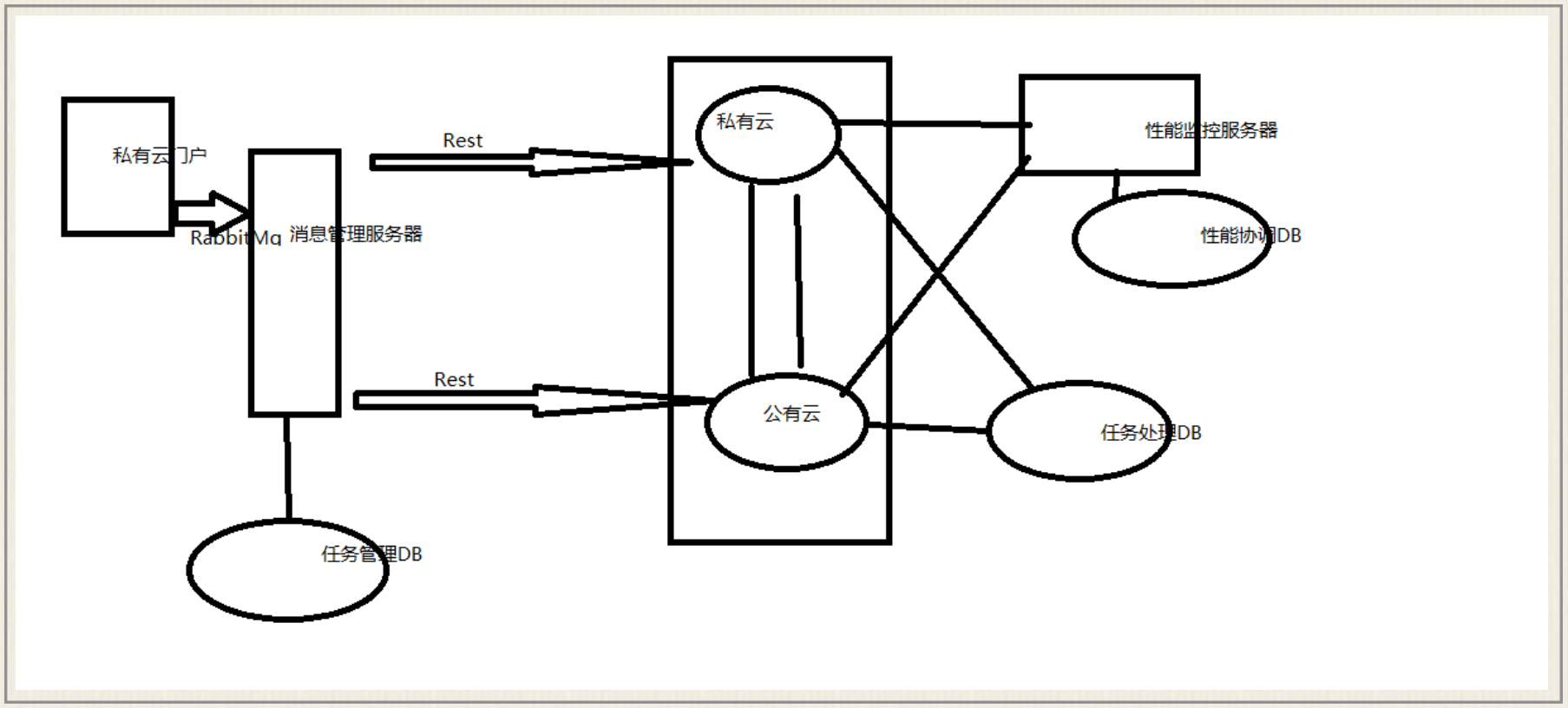
比如：

- 不同操纵系统之间的特点；
- 网络端口管理与分发；
- 哪些网络协议可以帮助我们更好的完成工作，监控虚拟机的时候是在虚机上加代理好还是用协议去控制；
- 硬件是否支持分布式，在扩展过程中对于.net C#的兼容怎么样；
- 什么时候使用多线程，在把线程交给程序调度的时候我们怎么控制和捕捉线程的异常；

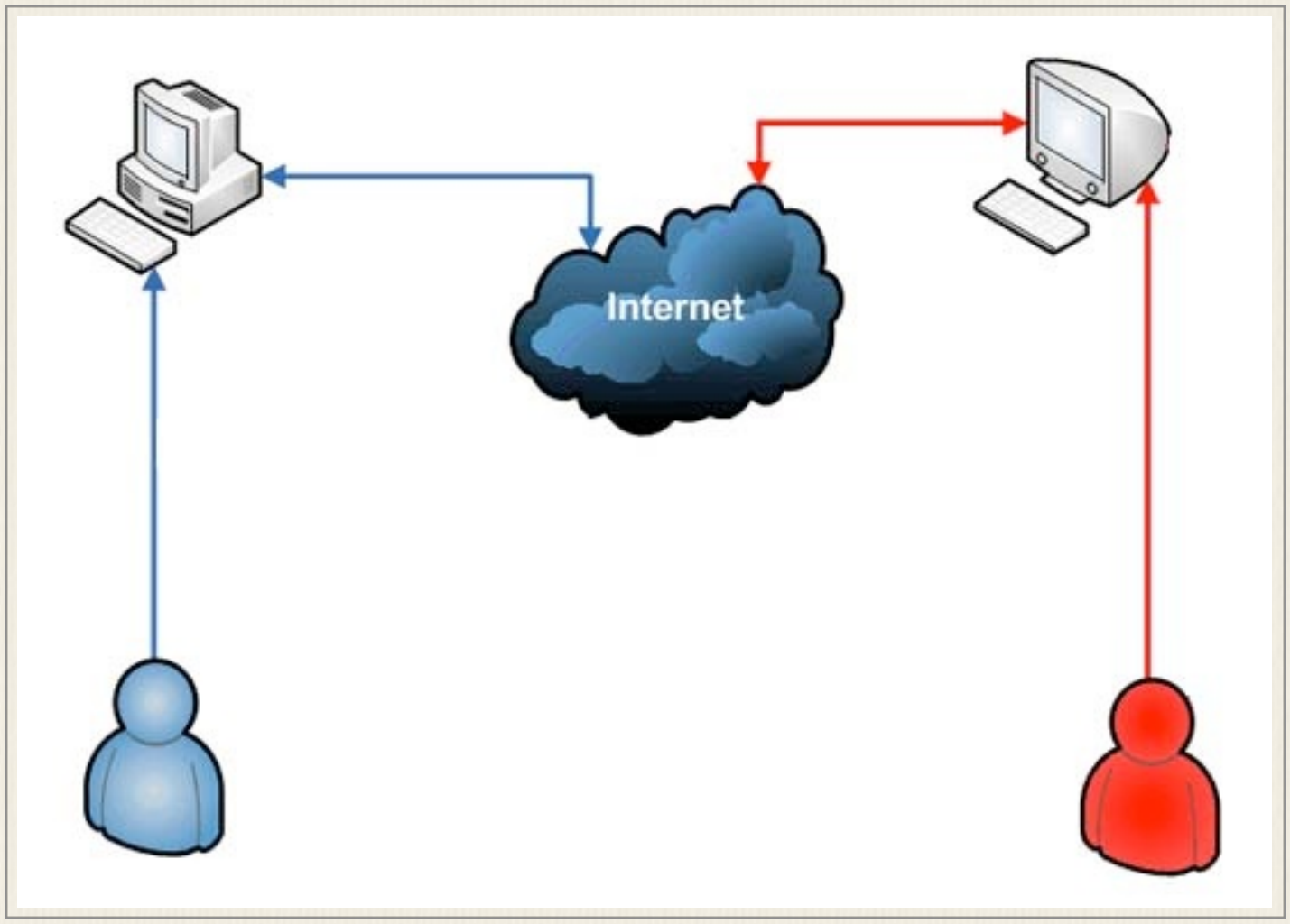
- 日志系统对于整个分散的系统是多么的重要；
- 何时使用关系数据库，什么时候使用Nosql；
- 消息队列用擅长的MSMQ还是RabbitMQ.
- 怎样有效的和其他部门的同事沟通；
- 用什么样的方式去有效调度不同语言开发的系统；
- 测试用例对于大系统从零散到完整是多么的重要；
- 系统标准，代码原则对于后期的维护余扩展是多么的重要；
- 等；

项目简介

首先项目详细内容不便多说，简答的说，就是为国内某大型厂商建立一套协调其自身搭建的私有云以及其购买的公有云的一套系统。说牛X一点就是：一套混合云系统。



使用Restful

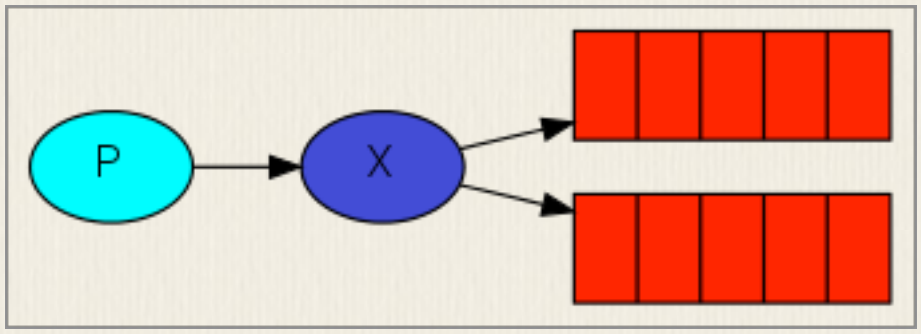


这是在构建完整个系统最大的收获，之前使用web api的经验只是为电商系统的移动终端提供数据交互的接口，但是在这次项目之后发现Rest接口的不仅作为我们系统向外部系统提供交互的方式，同时在一些 开源工具其暴露出来的接口也是基于rest的，可见全世界的程序员对于json对于rest有多么的喜爱；

为什么装上软件配置完成之后使用不了

之前的开发经验由于使用的都是微软的技术包括组建，工具。但是在项目中使用一些开源工具之后，配置成功之后却总也跑不起来，同时由于开源工具已将 Exception封装起来，我们很难知道具体是什么样的问题，有的时候调试好久还是跑不起来，很沮丧也很懊恼，结果最后发现是由于公司IT只是将常用的 端口打开，其他的都干掉了，如果申请开放端口的话还要走流程，于是对于开发人员有时候有一台外网的开发虚拟机也是相当的有必要的。

使用RabbitMq



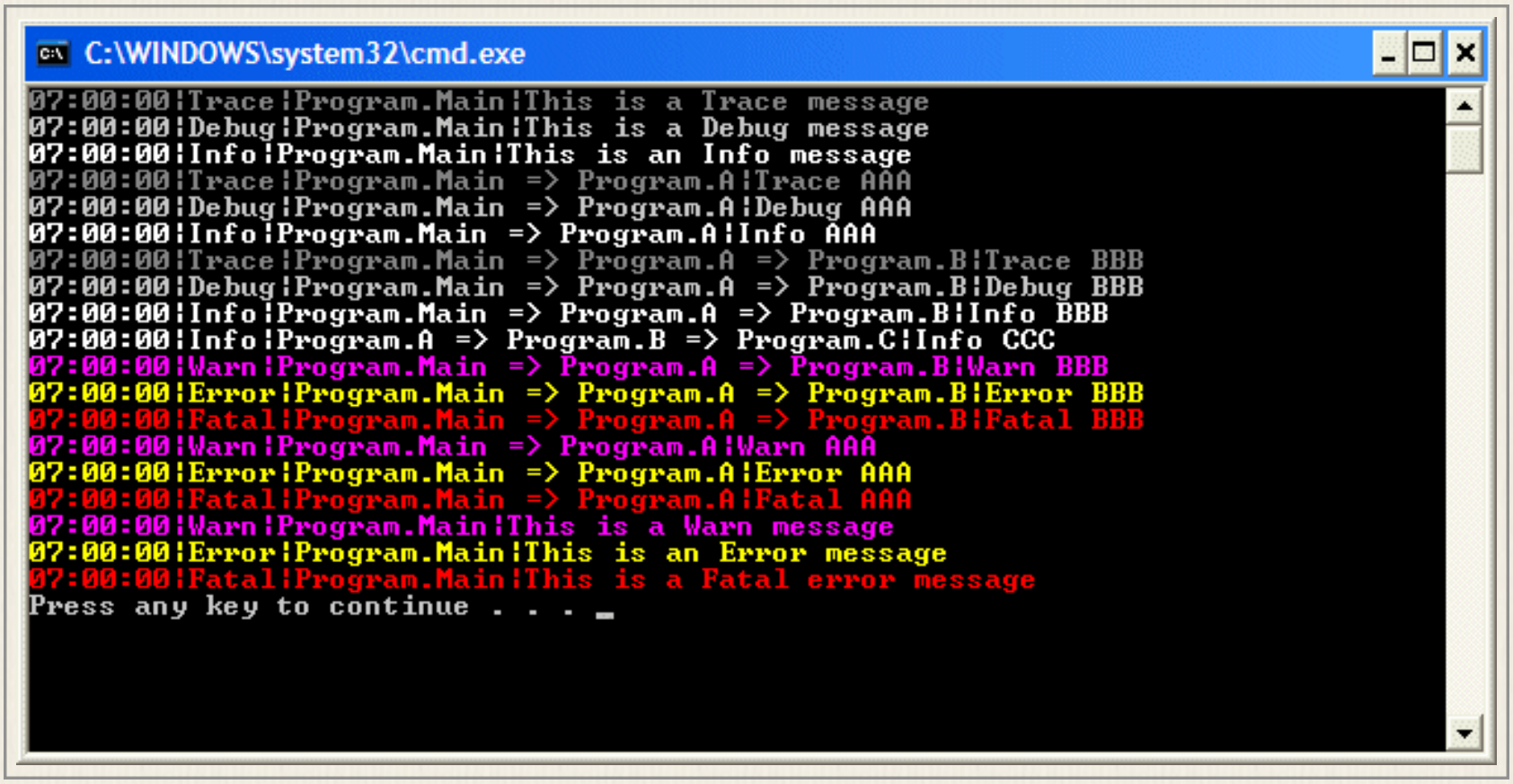
个人是非常的喜欢使用Mq的，之前做电商总喜欢在Application层下面放入一层Service，你可以不用但是总会强迫症似的不去不写。为 什么不用Msmq呢，原因是有很多了，简单点就是rabbit要比Msmq的协议更加高级，支持的处理功能也更加丰富，最重要的原因是Rabbit在开源 语言使用上是占领先地位的，而且我们的系统又要嫁接太多的开源语言系统，最后只能适配他们喽。

之前知道Mq在企业系统间数据交互使用频繁，不但能有效的划分层次，解耦依赖，同时数据交互方式上也相当的便捷。总会有消息没有被消费者使用，那我们就需要程序异步的去处理这个消息队列了。

Redis

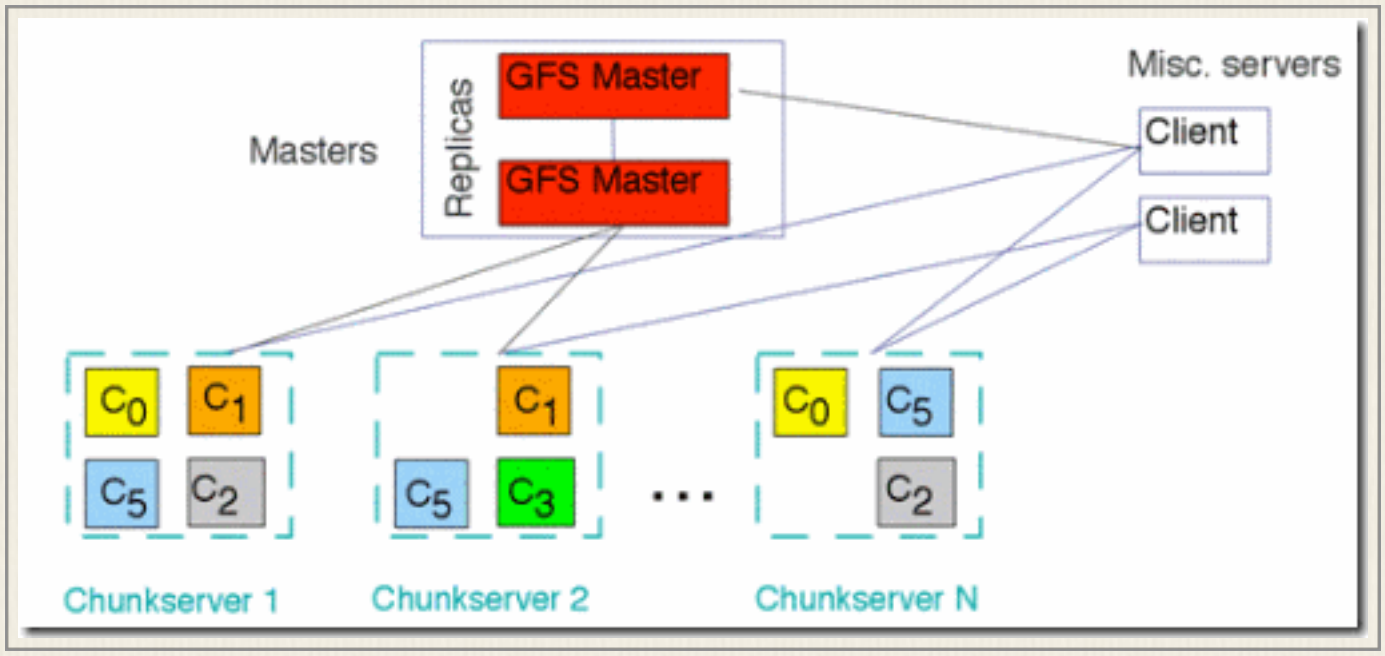
系统中使用了三种数据存储，MySql，SqlServer，Redis，当然前两种适用于开源和C#，而Redis的使用则是为了那些总是难以找 到有效关系和依赖的数据，比如之前只是知道Reids可以作为数据的存储，可以分布式，可以主从复制，但是在这次开发之后更真真的发现Reids或者 Nosql对于一个数据规则难以掌握，数据量大的系统是多么的重要，因为有的时候一批的Json串过来之后，难以有效的挖出里面的关系与逻辑，索性就一次 性将他们放入Redis中吧，使用时再反序列吧。同时建立读写分离的原则，我们主要将读放在了Redis里面，写到了Mysql，并通过Mysql的触发 器实现服务器段数据的主从复制同步。

日志系统



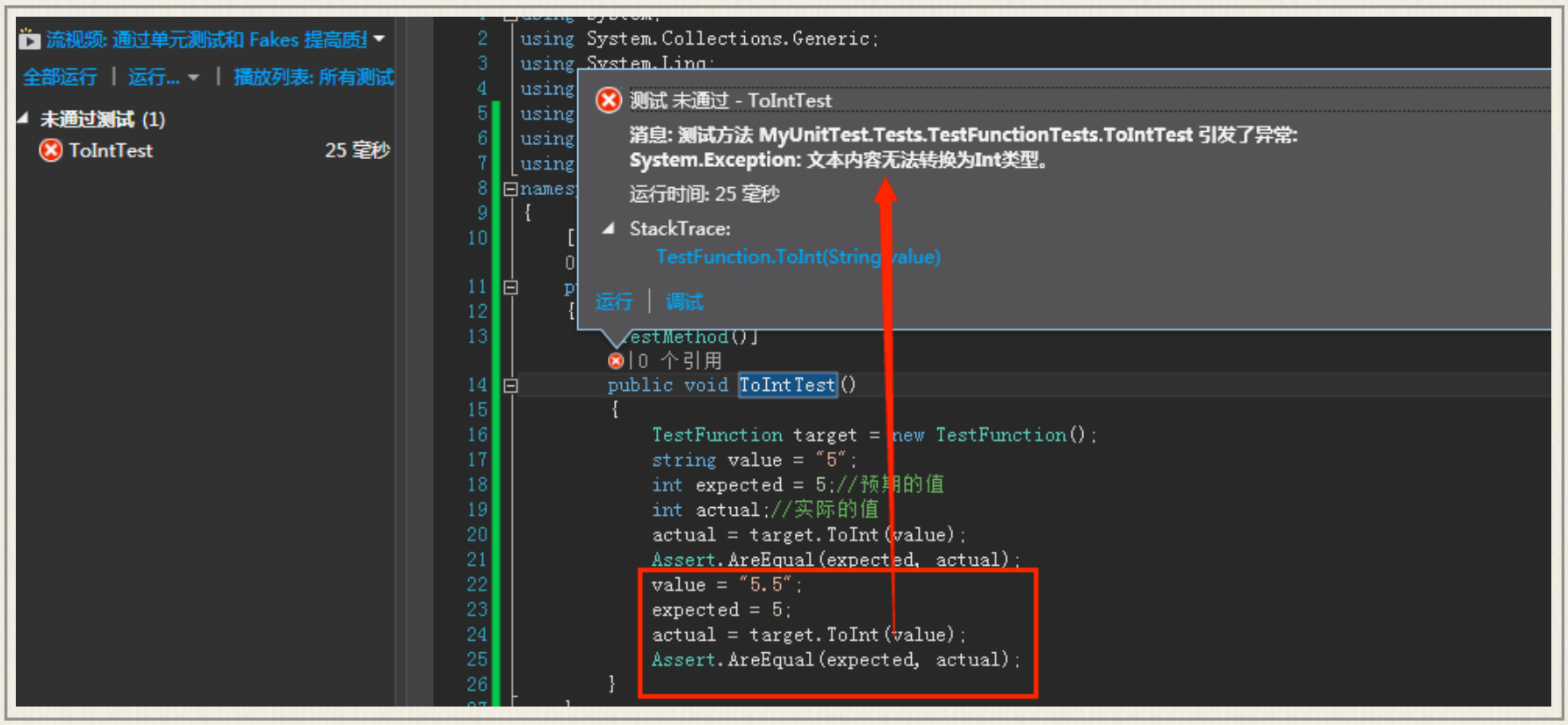
之前我们的单一系统的时候，比如只是简单的3层架构的话，我们通过Debug可以从头debug到数据库，每一步都是掌握在手底下，每一步都尽收眼底。可 是对于这一个层次太深，组建调用较多，同时又是多线程的系统来说，挖到雷的机会，时间，成本都是要考虑的。于是有效的使用日志组件，有效的在代码中埋雷就 显的尤为迫切和必要，能够更好的帮助我们找到问题所在。

组件式开发



之前的简单分层系统我们通过Svn或其他的代码管理工具，每次提交都可以Merge看的到，但是当系统庞杂同时系统独立性很强的时候，分组建，分模块开发 就显得很重要。因为不想浪费大家一起Merge的时间，我们习惯性每个人有自己的Branch每周2的时候提交代码，大家一起参与，这样减少了好多因为代 码管理浪费的时间。

测试用例



之前小的系统使用测试用例基本就是装B用的，本来小小的系统整套流程脑子一想就可以知道怎么做啦，为什么还要浪费时间。可是在这次开发中充分理解了 测试用例的重要性。比如我需要你给我提供多台服务器的监控数据包括CPU信息，IO信息，NEt信息等等，但是你还没有想到怎么样去抓取虚拟信息，不能因 为你的问题去影响其他人的进度的，最好的方式从使用者角度获知他希望使用什么样的数据，为其建立API，同时为API建立测试用例并保证测试稳定。而后期 我有了监控虚机的方式之后我在建立对应的适配方法适配到对应的API上。

所以首先肯定要保证API的稳定，因为他之上的东西已经稳定了吗，你只好辛苦啦，有效的测试用例可以帮助我们更好的剥离项目逻辑与协调组件系统。

编码原则

这个主要是每周有时间大家一起参与Code Review，由于开发人员的能力不同资历不同，所以总会在代码的编写上和建立出现太多的不统一。比如命名啦，变量声明啦，有的时候会发现刚毕业的小朋友 会将好多的私有变量放在类的顶部，同时一个类里写太多的方法，而且有的方法好长，还没有注释。于是有的时候你想了解一个方法的真正含义，要鼠标各种滚动， 到变量声明去了解真正用途，好烦的。

有的时候代码的职责不明确，总是瀑布的思想方式去写代码，比如我们两个功能：一个是发送API请求建立虚拟机，另一个是在虚拟机建立成功时

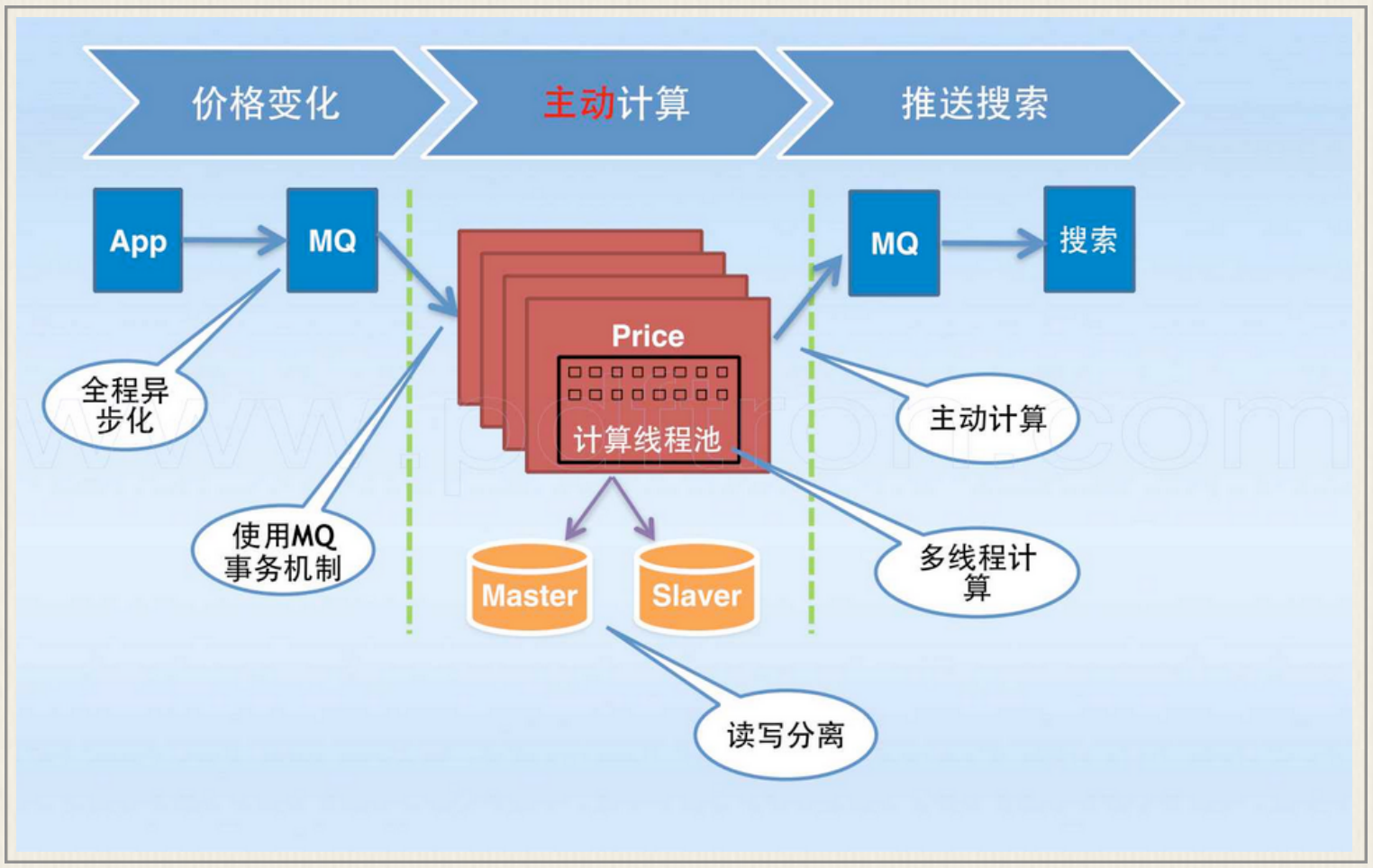
候将操作 Log写入db。他们习惯性的将写DB的逻辑放在了发送HTTPRequst的方法里面，这完全是两个逻辑。另一个问题是由于创建虚拟机是需要时间的，同 时尽管虚拟机操作成功有可能你写DB的时候网络原因DB失败了。我认为这应该是个原子的操作，两者的状态必须统一，就像是你在手机充值的时候显示银行卡扣 金额成功，可是手机充值是出现问题，钱不是白花了吗。所以在这些有特殊逻辑的地方要建立特殊的统一的机制，不能每个人有各自的实现。

之后就是沟通了

由于项目涉及到多个项目组，我们并不是同一个部门，相互也不熟悉，所以沟通上就会有一些需要注意的。首先要了解“对手”，主要是因为如果对方是个技 术高手，你不能像个白痴小孩，要有所准备，最起码知道他们用什么开发语言，他们需要关注的业务逻辑，等等，不能让他们得到你是个菜鸟的结论。

由于口头的好多东西可能是没有经过检验的东西，所以前几次达成的协议我们只是做个参考，需要多次沟通之后才能确定结果，比如我们的项目中我们需要和 Python组Java组协调消息接口，消息格式的时候。你要知道协调RabbitMQ时候我们需要定义下交互的Exchange，queue name 或者RoteKey等等，同时由于消息格式比较大，需要定义一些关键字或者预设字段的话，需要发邮件进行确认与沟通，避免开发过程中产生误会影响完成的功 能返工。

总之这次搭架子的过程收获很多，一时半会也不能想的全面，以后慢慢聊，由于是第一次资历尚浅，好多的技术选型，问题考虑可能不成熟，也希望大家知道更多的能够纠错指导。



下面就说一些我们在架构中使用的一些东西：

开发语言：C#，java，Python；

数据存储：缓存，文件（xml）,MSsql，Mysql，Redis；

数据交互：rest,json,RabbitMq;

操作系统：ubuntu,windows;

虚拟机监控：zabbix；

搜索：solr;

多线程，多层架构，模块式开发，组件式开发；

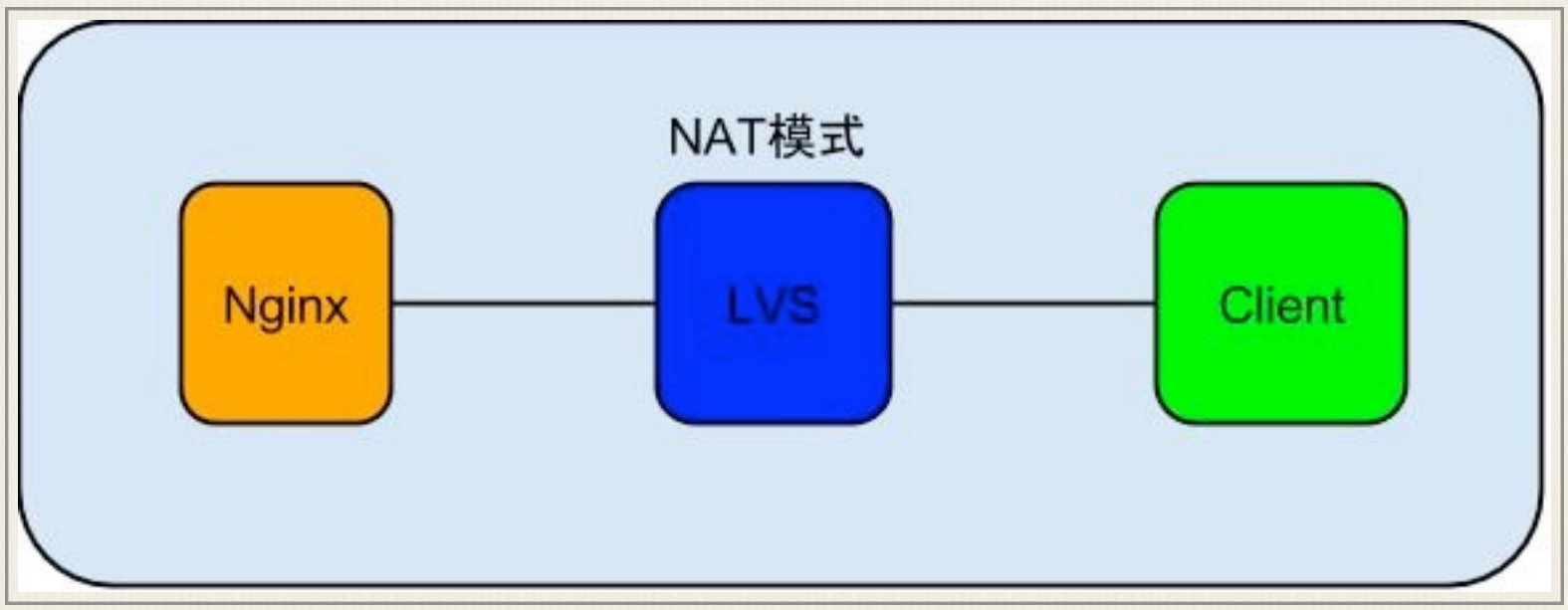
原文链接：http://www.cnblogs.com/xiguain/p/3839944.html

又见KeepAlive

作者：李成栋

我为什么要谈KeepAlive

最近工作中遇到一个问题，想把它记录下来，场景是这样的：



从上图可以看出，用户通过Client访问的是LVS的VIP， VIP后端挂载的RealServer是Nginx服务器。 Client可以是浏览器也可以是一个客户端程序。一般情况下，这种架构不会出现问题，但是如果Client端把请求发送给Nginx，Nginx的后端需要一段时间才能返回结果，超过1分30秒就会有问题，使用LVS作为负载均衡设备看到的现象就是1分30秒之后， Client和Nginx链接被断开，没有数据返回。原因是LVS默认保持TCP的Session为90s，超过90s没有TCP报文在链接上传输，LVS就会给两端发送RESET报文断开链接。LVS这么做的原因相信大家都知道一二，我所知道的原因主要有两点：

1.节省负载均衡设备资源，每一个TCP/UDP的链接都会在负载均衡设备上创建一个Session的结构，

链接如果一直不断开，这种Session 结构信息最终会消耗掉所有的资源，所以必须释放掉。

2.另外释放掉能保护后端的资源，如果攻击者通过空链接，链接到Nginx上，如果Nginx没有做合适

的保护，Nginx会因为链接数过多而无法提供服务。

这种问题不只是在LVS上有，之前在商用负载均衡设备F5上遇到过同样的问题，F5的Session断开方式和LVS有点区别，F5不会主动发送RESET给链接的两端，Session消失之后，当链接中一方再次发送报文时会接收到F5的RESET, 之后的现象是再次发送报文的一端TCP链接状态已经断开，而另外一端却还是ESTABLISH状态。

知道是负载均衡设备原因之后，第一反应就是通过开启KeepAlive来解决。到此这个问题应该是结束了，但是我发现过一段时间总又有人提起KeepAlive的问题，甚至发现由于KeepAlive的理解不正确浪费了很多资源，原本能使用LVS的应用放在了公网下沉区，或者换成了商用F5设 备(F5设备的Session断开时间要长一点，默认应该是5分钟)。所以我决定把我知道的KeepAlive知识点写篇博客分享出来。

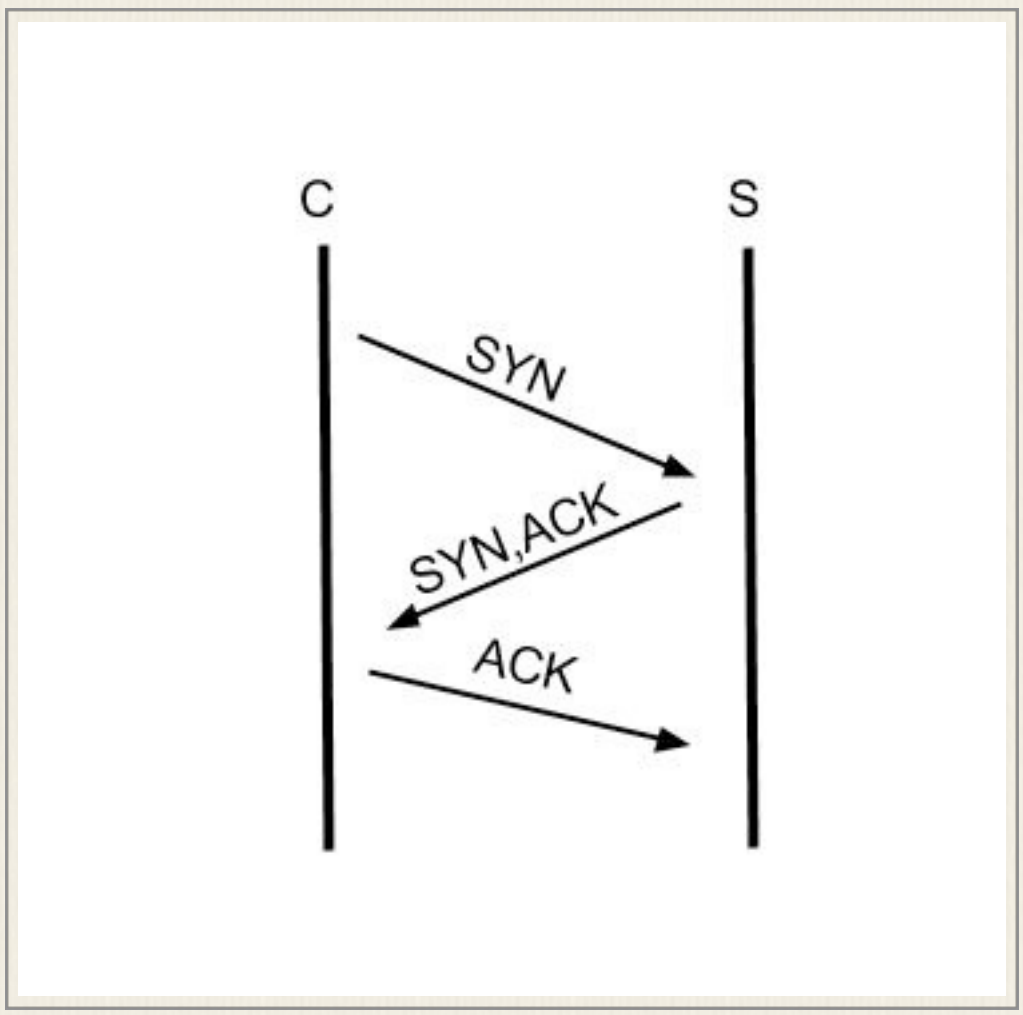
为什么要有KeepAlive?

在谈KeepAlive之前，我们先来了解下简单TCP知识(知识很简单，高手直接忽略)。首先要明确的是在TCP层是没有“请求”一说的，经常听到在TCP层发送一个请求，这种说法是错误的。TCP是一种通信的方式，“请求”一词是事务上的概念，HTTP协议是一种事务协议，如果说发送一个HTTP请 求，这种说法就没有问题。也经常听到面试官反馈有些面试运维的同学，基本的TCP三次握手的概念不清楚，面试官问TCP是如何建立链接，面试者上来就说， 假如我是客户端我发送一个请求给服务端，服务端发送一个请求给我。。。这种一听就知道对TCP基本概念不清楚。下面是我通过

wireshark抓取的一个 TCP建立握手的过程。（命令行基本上用TCPdump, 后面我们还会用这张图说明问题）：

	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.22.2.85	10.147.208.49	TCP	74	58936 > http [SYN] Seq=0 Win=5840 Len=0 MSS=1460 SACK_PERM=1 TSval=38492
2	0.000979	10.147.208.49	172.22.2.85	TCP	74	http > 58936 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1452 SACK_PERM=1
3	0.001013	172.22.2.85	10.147.208.49	TCP	54	58936 > http [ACK] Seq=1 Ack=1 Win=5840 Len=0
4	0.001065	172.22.2.85	10.147.208.49	HTTP	376	GET /tigo/traceparam/httpreq?fields=stat_date,trace_param_value&filter=s
5	0.002436	10.147.208.49	172.22.2.85	TCP	60	http > 58936 [ACK] Seq=1 Ack=323 Win=15544 Len=0
6	90.001903	10.147.208.49	172.22.2.85	TCP	60	http > 58936 [RST] Seq=1 Win=0 Len=0

现在我看只要看前3行，这就是TCP三次握手的完整建立过程，第一个报文SYN从发起方发出，第二个报文SYN,ACK是从被连接方发出，第三个报文ACK确认对方的SYN，ACK已经收到，如下图：



但是数据实际上并没有传输，请求是有数据的，第四个报文才是数据传输开始的过程，细心的读者应该能够发现wireshark把第四个报文解析成HTTP协 议，HTTP协议的GET方法和URI也解析出来，所以说TCP层是没有请求的概念，HTTP协议是事务性协议才有请求的概念，TCP报文承载HTTP协 议的请求(Request)和响应(Response)。

现在才是开始说明为什么要有KeepAlive。链接建立之后，如果应用程序或者上层协议一直不发送数据，或者隔很长时间才发送一次数据，当链接很久没有 数据报文传输时如何去确定对方还在线，到底是掉线了还是确实没有数据传输，链接还需不需要保持，这种情况在TCP协议设计中是需要考虑到的。TCP协议通 过一种巧妙的方式去解决这个问题，当超过一段时间之后，TCP自动发送一个数据为空的报文给对方，如果对方回应了这个报文，说明对方还在线，链接可以继续 保持，如果对方没有报文返回，并且重试了多次之后则认为链接丢失，没有必要保持链接。

如何开启KeepAlive

KeepAlive并不是默认开启的，在Linux系统上没有一个全局的选项去开启TCP的KeepAlive。需要开启KeepAlive的应用必须在TCP的socket中单独开启。Linux Kernel有三个选项影响到KeepAlive的行为：

1.net.ipv4.tcpkeepaliveintvl = 75
2.net.ipv4.tcpkeepaliveprobes = 9
3.net.ipv4.tcpkeepalivetime = 7200
tcpkeepalivetime的单位是秒，表示TCP链接在多少秒之后没有数据报文传输启动探测报文; tcpkeepaliveintvl单位是也秒,表示前一个探测报文和后一个探测报文之间的时间间隔，tcpkeepaliveprobes表示探测的次数。

TCP socket也有三个选项和内核对应，通过setsockopt系统调用针对单独的socket进行设置：

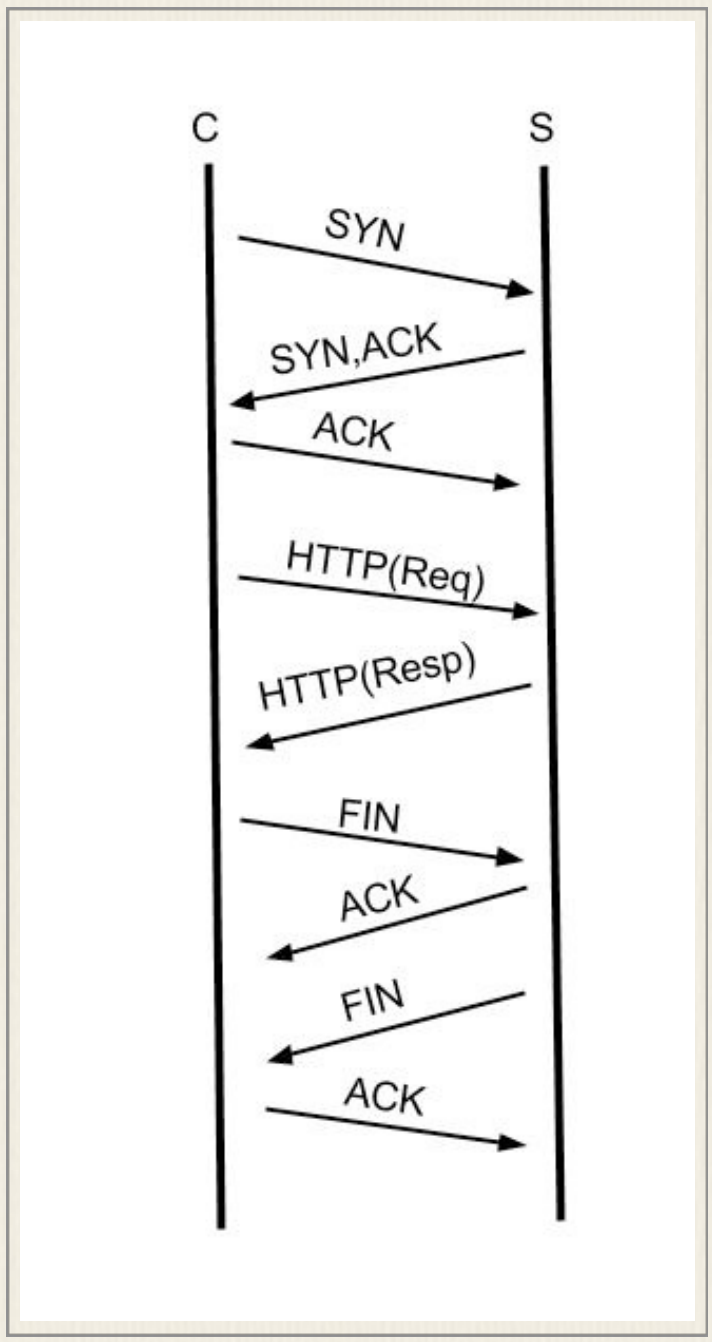
TCPKEEPCNT: 覆盖 tcpkeepaliveprobes
TCPKEEPIDLE: 覆盖 tcpkeepalivetime
TCPKEEPINTVL: 覆盖 tcpkeepalive_intvl

举个例子，以我的系统默认设置为例，kernel默认设置的tcpkeepalivetime是7200s, 如果我在应用程序中针对socket开启了

KeepAlive,然后设置的TCP_KEEPIDLE为60，那么TCP协议栈在发现TCP链接空闲了60s没有数据传输的时候就会发送第一个探测报文。

TCP KeepAlive和HTTP的Keep-Alive是一样的吗?

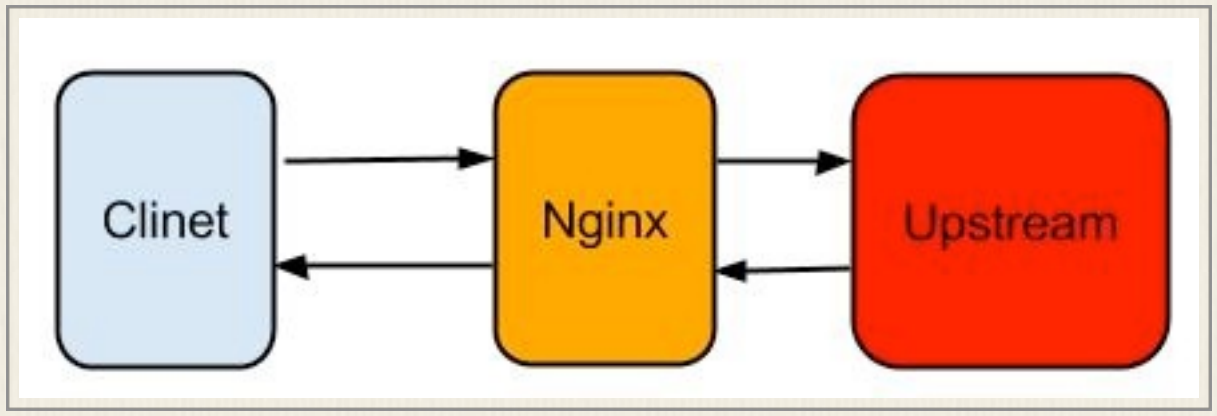
估计很多人乍看下这个问题才发现其实经常说的KeepAlive不是这么回事，实际上在没有特指是TCP还是HTTP层的KeepAlive，不能混为一 谈。TCP的KeepAlive和HTTP的Keep-Alive是完全不同的概念。TCP层的KeepAlive上面已经解释过了。 HTTP层的Keep-Alive是什么概念呢？ 在讲述TCP链接建立的时候，我画了一张三次握手的示意图，TCP在建立链接之后， HTTP协议使用TCP传输HTTP协议的请求(Request)和响应(Response)数据，一次完整的HTTP事务如下图:



各位看官请注意，这张图我简化了HTTP(Req)和HTTP(Resp)，实际上的请求和响应需要多个TCP报文。从图中可以发现一个完整的HTTP事 务，有链接的建立，请求的发送，响应接收，断开链接这四个过程,早期通过HTTP协议传输的数据以文本为主，一个请求可能就把所有要返回的数据取到，但是，现在要展现一张完整的页面需要很多个请求才能完成，如图片,JS,CSS等，如果每一个HTTP请求都需要新建并断开一个TCP，这个开销是完全没有 必要的，开启HTTP Keep-Alive之后，能复用已有的TCP链接，当前一个请求已经响应完毕，服务器端没有立即关闭TCP链接，而是等待一段时间接收浏览器端可能发送 过来的第二个请求，通常浏览器在第一个请求返回之后会立即发送第二个请求，如果某一时刻只能有一个链接，同一个TCP链接处理的请求越多，开启 KeepAlive能节省的TCP建立和关闭的消耗就越多。当然通常会启用多个链接去从服务器器上请求资源，但是开启了Keep-Alive之后，仍然能 加快资源的加载速度。HTTP/1.1之后默认开启Keep-Alive, 在HTTP的头域中增加Connection选项。当设置为Connection:keep-alive表示开启，设置为 Connection:close表示关闭。实际上HTTP的KeepAlive写法是Keep-Alive，跟TCP的KeepAlive写法上也有不 同。所以TCP KeepAlive和HTTP的Keep-Alive不是同一回事情。

Nginx的TCP KeepAlive如何设置

开篇提到我最近遇到的问题，Client发送一个请求到Nginx服务端，服务端需要经过一段时间的计算才会返回， 时间超过了LVS Session保持的90s,在服务端使用Tcpdump抓包,本地通过wireshark分析显示的结果如第二副图所示，第5条报文和最后一条报文之间的时间戳大概差了90s。在确定是LVS的Session保持时间到期的问题之后，我开始在寻找Nginx的TCP KeepAlive如何设置，最先找到的选项是keepalivetimeout,从同事那里得知keepalivetimeout的用法是当keepalivetimeout的值为0时表示关闭keepalive,当keepalivetimeout的值为一个正整数值时表示链接保持多少秒，于是把keepalivetimeout设置成75s,但是实际的测试结果表明并不生效。显然keepalivetimeout不能解决TCP层面的KeepAlive问题，实际上Nginx涉及到keepalive的选项还不少，Nginx通常的使用方式如下：



从TCP层面Nginx不仅要和Client关心KeepAlive,而且还要和Upstream关心KeepAlive, 同时从HTTP协议层面，Nginx需要和Client关心Keep-Alive,如果Upstream使用的HTTP协议，还要关心和Upstream 的Keep-Alive，总而言之，还比较复杂。所以搞清楚TCP层的KeepAlive和HTTP的Keep-Alive之后，就不会对于Nginx的 KeepAlive设置错。我当时解决这个问题时候不确定Nginx有配置TCP keepAlive的选项，于是我打开Ngnix的源代码，在源代码里面搜索TCP_KEEPIDLE,相关的代码如下：

```
519 #if (NGX_HAVE_KEEPALIVE_TUNABLE)
520
521         if (ls[i].keepidle) {
522             if (setsockopt(ls[i].fd, IPPROTO_TCP, TCP_KEEPIDLE,
523                            (const void *) &ls[i].keepidle, sizeof(int))
524                 == -1)
525             {
526                 ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
527                               "setsockopt(TCP_KEEPIDLE, %d) %V failed, ignored",
528                               ls[i].keepidle, &ls[i].addr_text);
529             }
530         }
```

从代码的上下文我发现TCP KeepAlive可以配置，所以我接着查找通过哪个选项配置，最后发现listen指令的so_keepalive选项能对TCP socket 进行KeepAlive的配置。

```
so_keepalive=on|off|[keepidle]:[keepintvl]:[keepcnt]
on表示开启
off表示关闭
有些系统提供跟精确的控制，比如linux:
    keepidle表示等待时间，keepintvl表示探测报的发送间隔，keepcnt表示探测报文发送的次数。
```

以上三个参数只能使用一个，不能同时使用， 比如sokeepalive=on, sokeepalive=off或者sokeepalive=30s::(表示等待30s没有数据报文发送探测报文)。通过设置listen 80,sokeepalive=60s::之后成功解决Nginx在LVS保持长链接的问题，避免了使用其他高成本的方案。在商用负载设备上如果遇到类似的问题同样也可以通过这种方式解决。

参考资料

《TCP/IP协议详解VOL1》--
强烈建议对于网络基本知识不清楚同学有空去看下。

http://tldp.org/HOWTO/html_single/TCP-Keepalive-HOWTO/#overview

http://nginx.org/en/docs/http/ngx_http_core_module.html

Nginx Source code: https://github.com/alibaba/tengine

原文链接：http://blog.sina.com.cn/s/blog_e59371cc0102ux5w.html

如何理解 TCP/IP, SPDY, WebSocket 三者之间的关系?

作者：用心阁

按照OSI网络分层模型，IP是网络层协议，TCP是传输层协议，而HTTP是应用层的协议。在这三者之间，SPDY和WebSocket都是与HTTP相关的协议，而TCP是HTTP底层的协议。

一、HTTP的不足

HTTP协议经过多年的使用，发现了一些不足，主要是性能方面的，包括：

- HTTP的连接问题，HTTP客户端和服务器之间的交互是采用请求/应答模式，在客户端请求时，会建立一个HTTP连接，然后发送请求消息，服务端给出应答消息，然后连接就关闭了。（后来的HTTP1.1支持持久连接）

- 因为TCP连接的建立过程是有开销的，如果使用了SSL/TLS开销就更大。

- 在浏览器里，一个网页包含许多资源，包括HTML，CSS，JavaScript，图片等等，这样在加载一个网页时要同时打开连接到同一服务器的多个连接。

- HTTP消息头问题，现在的客户端会发送大量的HTTP消息头，由于一个网页可能需要50-100个请求，就会有相当大的消息头的数据量。

- HTTP通信方式问题，HTTP的请求/应答方式的会话都是客户端发起的，缺乏服务器通知客户端的机制，在需要通知的场景，如聊天室，游戏，客户端应用需要不断地轮询服务器。

而SPDY和WebSocket是从不同的角度来解决这些不足中的一部分。除了这两个技术，还有其他技术也在针对这些不足提出改进。

二、SPDY

SPDY的主要目的是减少50%以上的页面加载时间，但是呢不增加部署的复杂性，不影响客户端和服务端的Web应用，只需要浏览器和Web服务器支持SPDY。主要有以下几点：

- 多路复用，一个TCP连接上同时跑多个HTTP请求。请求可设定优先级。
- 去除不需要的HTTP头，压缩HTTP头，以减少需要的网络带宽。
- 使用了SSL作为传输协议提供数据安全。
- 对传输的数据使用gzip进行压缩
- 提供服务方发起通信，并向客户端推送数据的机制。

实质上，SPDY就是想不影响HTTP语义的情况下，替换HTTP底层传输的协议来加快页面加载时间。

SPDY的解决办法就是设计了一个会话层协议--
帧协议，解决多路复用，优先级等问题，然后在其上实现了HTTP的语义。

三、WebSocket

WebSocket则提供使用一个TCP连接进行双向通讯的机制，包括网络协议和API，以取代网页和服务器采用HTTP轮询进行双向通讯的机制。

本质上来说，WebSocket是不限于HTTP协议的，但是由于现存大量的HTTP基础设施，代理，过滤，身份认证等等，WebSocket借用HTTP和HTTPS的端口。

由于使用HTTP的端口，因此TCP连接建立后的握手消息是基于HTTP的，由服务器判断这是一个HTTP协议，还是WebSocket协议。 WebSocket连接除了建立和关闭时的握手，数据传输和HTTP没丁点关系了。

WebSocket也有自己一套帧协议。

四、SPDY和WebSocket的关系

SPDY和WebSocket的关系比较复杂。

1. 补充关系，二者侧重点不同。SPDY更侧重于给Web页面的加载提速，而WebSocket更强调为Web应用提供一种双向的通讯机制以及API。

2. 竞争关系，二者解决的问题有交集，比如在服务器推送上SPDY和WebSocket都提供了方案。

3. 承 载关系，试想，如果SPDY的标准化早于WebSocket，WebSocket完全可以侧重于API，利用SPDY的帧机制和多路复用机制实现该 API。 Google提出草案，说WebSocket可以跑在SPDY之上。WebSocket的连接建立在SPDY的流之上，将WebSocket的帧映射到SPDY的帧上。

4. 融合关系，如微软在HTTP Speed+Mobility中所做的。

五、题外话

1. HTTP Speed+Mobility

还有一个有趣的技术叫做HTTP Speed+Mobility，和SPDY一样都是HTTP 2.0标准的竞争者，HTTP Speed+Mobili-

ty来自微软。HTTP SM借鉴了SPDY和WebSock-
et的协议，将二者揉为一体，又有所取舍。

HTTP SM的设计原则包括：

- 保留HTTP的语义，这一点和SPDY一致，但也正应如此，抛弃了SPDY里的ServerPush。
- 遵守分层的网络架构，TCP能做的，HTTP SM不做，因此去除了SPDY的流控。
- 使用现有标准，因此使用HTTP/1.1 Upgrade head-
er机制，借用了WebSocket的握手机制和帧格式（RFC6455）。
- 客户端掌握内容的控制，因此不强制使用压缩和SSL/TLS。
- 考虑到网络的费用和电力，这点考虑到了移动设备以及物联网，提供了Credit Control机制。

HTTP SM分以下几层：

- 会话层和帧协议，这部分取自WebSock-
et协议。包括握手机制，以及帧格式。
- 流层（包括多路复用），这部分主要借鉴SPDY，包括多路复用，流优先级，但增加了Credit Control。这部分作为 WebSocket协议的扩展。
- HTTP层，在流层上实现HTTP语义，这部分也借鉴自SPDY。

2. Network-Friendly HTTP

NF是HTTP 2.0候选方案之一，主要提出以下改进：

- 对HTTP头的名称进行二进制编码
- 对通用HTTP头进行分组
- 请求/应答的多路复用

- 分层模型

NF同样定义了帧和流，

3. WAKA

WAKA也是HTTP 2.0候选方案之一，是HTTP协议原作者Roy Fielding提出的一个提案。

WAKA支持多路复用，支持优先级。WAKA提出了两个新的HTTP方法，RENDER和MONITOR。

原文链接：http://www.zhihu.com/question/20097129/answer/15017791

iOS应用程序的生命周期

作者：EverNight

对于iOS应用程序，关键的是要知道你的应用程序是否正在前台或后台运行。由于系统资源在iOS 设备上较为有限，一个应用程序必须在后台与前台有不同的行为。操作系统也会限制你的应用程序在后台的运行，以提高电池寿命，并提高用户与前台应用程序的体 验。当应用程序在前台和后台之间切换时，操作系统将会通知您的应用程序。你可以通过这些通知来修改你的应用程序的行为。

当你的应用程序在前台活动时，系统会发送触摸事件给它进行处理。在UIKit的基础设施做了大部分的事件传递给你的自定义对象工作。所有您需要做的是覆盖在相应的对象的方法来处理这 些事件。对于控件，UIKit会通过处理你的触摸事件，或者其他一些有趣的事情发生时调用您的自定义代码，比如当文本字段中的值更改。

1：应用程序的状态

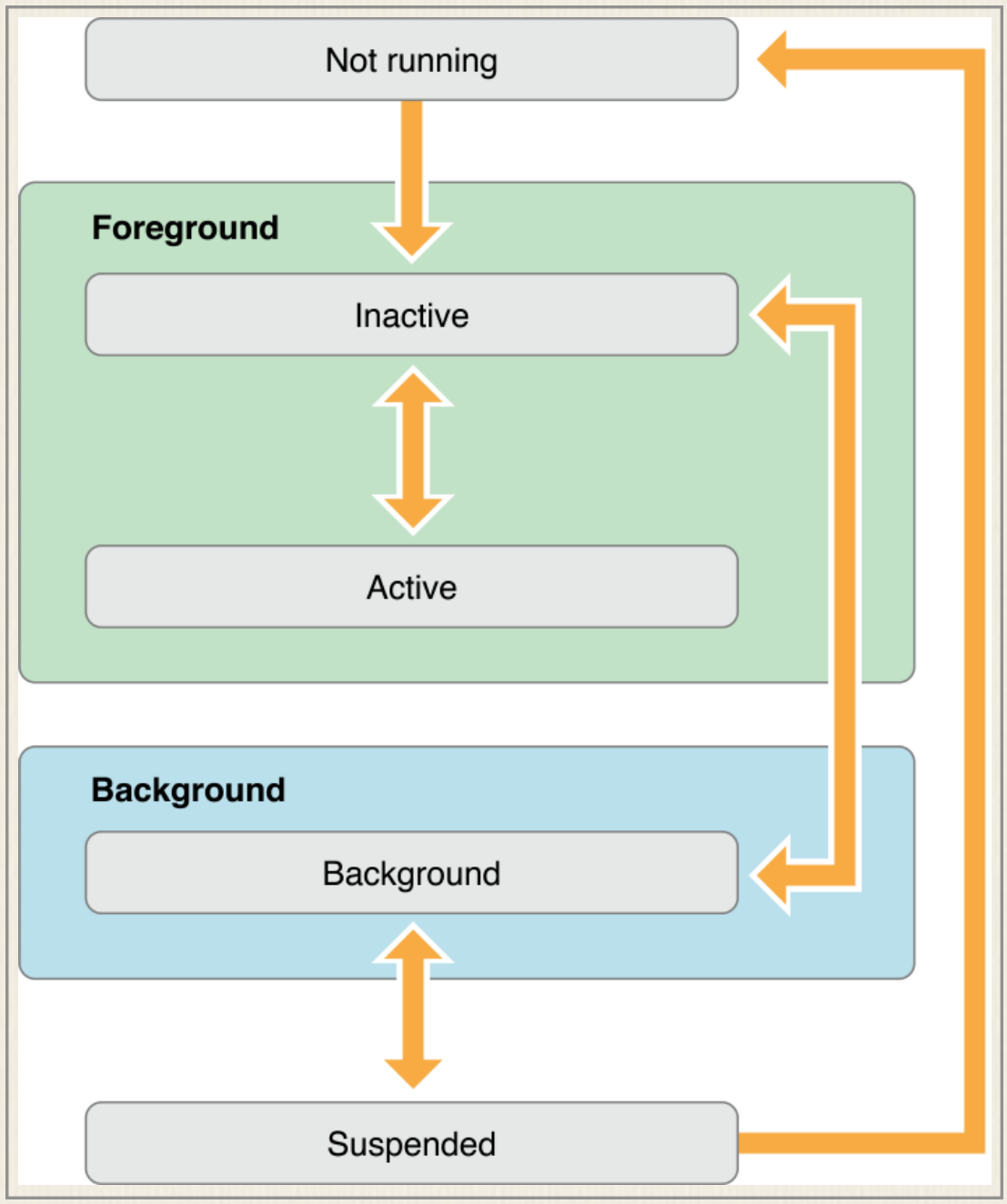
Not running未运行：程序没启动。

Inactive未激活：程序在前台运行，不过没有接收到事件。在没有事件处理情况下程序通常停留在这个状态。

Active激活：程序在前台运行而且接收到了事件。这也是前台的一个正常的模式。

Backgroud后台：程序在后台而且能执行代码，大多数程序进入这个状态后会在在这个状态上停留一会。时间到之后会进入挂起状态(Suspended)。有的程序经过特殊的请求后可以长期处于Backgroud状态。

Suspended挂起：程序在后台不能执行代码。系统会自动把程序变成这个状态而且不会发出通知。当挂起时，程序还是停留在内存中的，当系统内存低时，系统就把挂起的程序清除掉，为前台程序提供更多的内存。



2：各个程序运行状态时代理的回调

①告诉代理进程启动但还没进入状态保存

```
1 - (BOOL)application:(UIApplication *)application willFinishLaunchingWithOptions:(NSDictionary *)launchOptions
2 {
3     NSLog(@"①告诉代理进程启动但还没进入状态保存");
4     return YES;
```

```
5 }
```

②告诉代理启动基本完成程序准备开始运行

```
1 - (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
2 {
3     self.window = [[UIWindow alloc] initWithFrame:[[UIScreen main-
Screen] bounds]];
4
5     NSLog(@"②告诉代理启动基本完成程序准备开始运行");
6
7     // Override point for customization after application launch.
8
9     self.window.backgroundColor = [UIColor whiteColor];
10     [self.window makeKeyAndVisible];
11     return YES;
12 }
```

③当应用程序将要入非活动状态执行，在此期间，应用程序不接收消息或事件，比如来电话

```
1 - (void)applicationWillResignActive:(UIApplication *)application
2 {
3     // Sent when the application is about to move from active to inac-
tive state. This can occur for certain types of temporary interruptions (such
```

as an incoming phone call or SMS message) or when the user quits the application and it begins the transition to the background state.

4 // Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL ES frame rates. Games should use this method to pause the game.

*5 NSLog(@"*③当应用程序将要入非活动状态执行，在此期间，应用程序不接收消息或事件，比如来电话*");*

6 }

④当应用程序进入活动状态执行

*1 - (void)applicationDidBecomeActive:(UIApplication *)application*

2 {

3 // Restart any tasks that were paused (or not yet started) while the application was inactive. If the application was previously in the background, optionally refresh the user interface.

*4 NSLog(@"*④当应用程序进入活动状态执行*");*

5 }

⑤当程序被推送到后台的时候调用。所以要设置后台继续运行，则在这个函数里面设置即可

*1 - (void)applicationDidEnterBackground:(UIApplication *)application*

2 {

3 // Use this method to release shared resources, save user data, invalidate timers, and store enough application state information to restore your application to its current state in case it is terminated later.

```
4    // If your application supports background execution, this method is called instead of applicationWillTerminate: when the user quits.
5    NSLog(@"⑤当程序被推送到后台的时候调用");
6
7    [application beginBackgroundTaskWithExpirationHandler:^{
8
9        NSLog(@"begin Background Task With Expiration Handler");
10
11   }];
12 }
```

⑥当程序从后台将要重新回到前台时候调用

```
1 - (void)applicationWillEnterForeground:(UIApplication *)application
2 {
3    // Called as part of the transition from the background to the inactive state; here you can undo many of the changes made on entering the background.
4    NSLog(@"⑥当程序从后台将要重新回到前台时候调用");
5 }
```

⑦当程序将要退出是被调用，通常是用来保存数据和一些退出前的清理工作。这个需要要设置UIApplicationExitsOnSuspend的键值

```
1 - (void)applicationWillTerminate:(UIApplication *)application
2 {
3    // Called when the application is about to terminate. Save data if appropriate. See also applicationDidEnterBackground:.
```

```
4     NSLog(@"⑦当程序将要退出是被调用");
5 }
```

⑧当程序载入后执行

```
1 - (void)applicationDidFinishLaunching:(UIApplication *)application
2 {
3     NSLog(@"⑧当程序载入后执行");
4 }
```

程序启动时:

2014-07-01 15:55:14.706 LifeCycle[5845:60b] ①告诉代理进程启动但还没进入状态保存

2014-07-01 15:55:14.708 LifeCycle[5845:60b] ②告诉代理启动基本完成程序准备开始运行

2014-07-01 15:55:14.709 LifeCycle[5845:60b] ④当应用程序进入活动状态执行

按下Home键返回主界面:

2014-07-01 15:56:11.756 LifeCycle[5845:60b] ③当应用程序将要入非活动状态执行

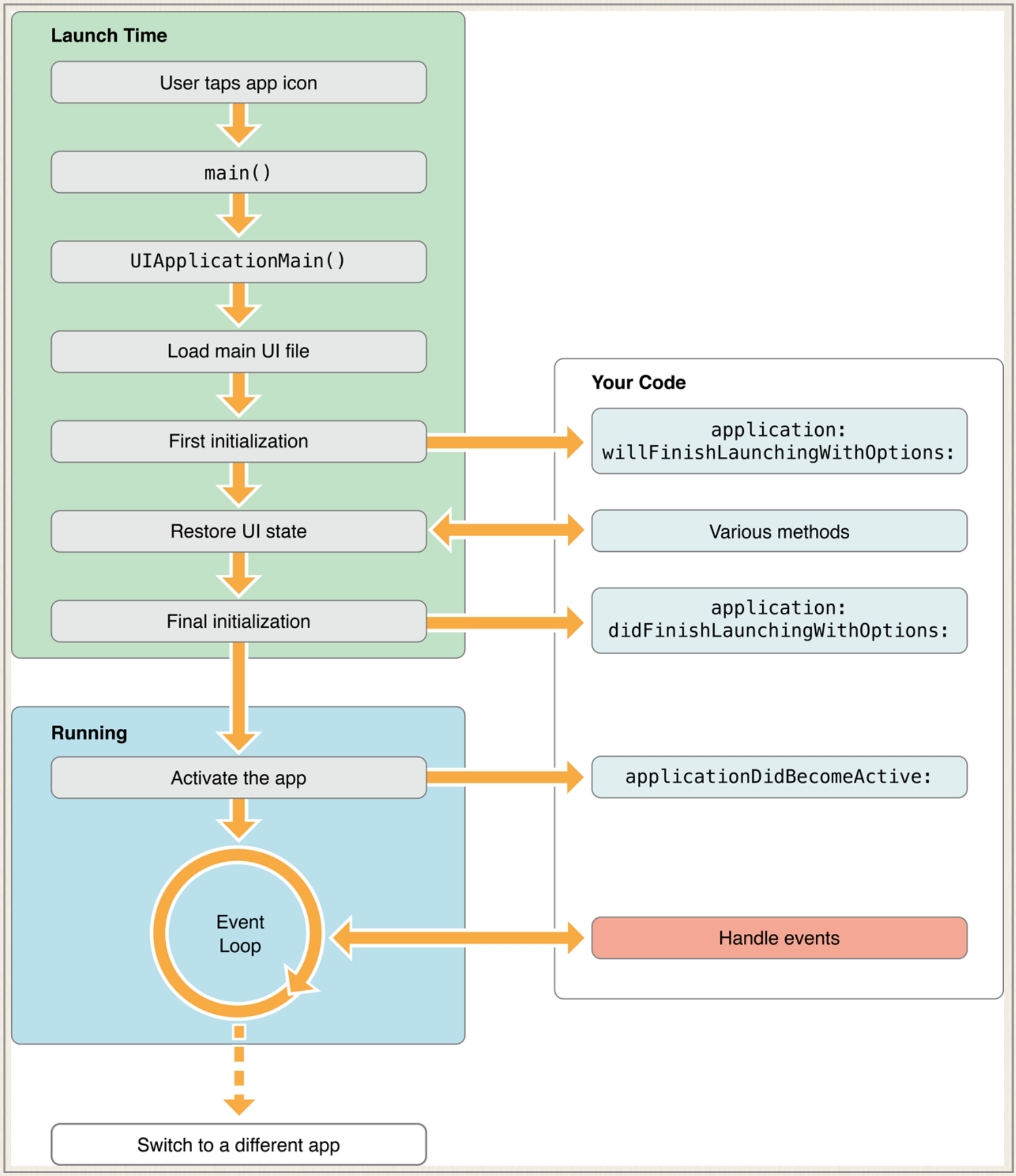
2014-07-01 15:56:11.814 LifeCycle[5845:60b] ⑤当程序被推送到后台的时候调用

再次打开程序:

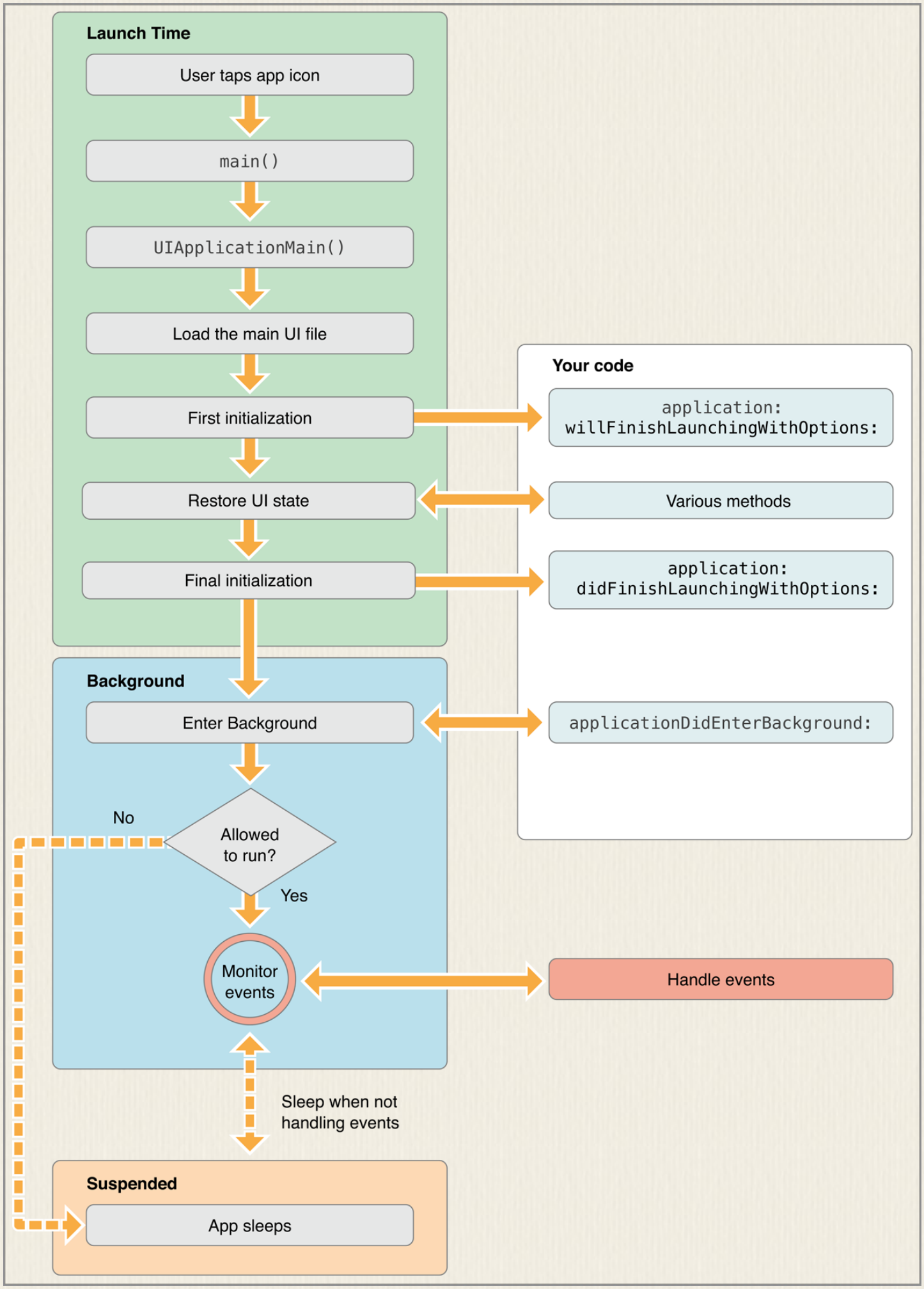
2014-07-01 15:57:19.200 LifeCycle[5845:60b] ⑥当程序从后台将要重新回到前台时候调用

2014-07-01 15:57:19.201 LifeCycle[5845:60b] ④当应用程序进入活动状态执行

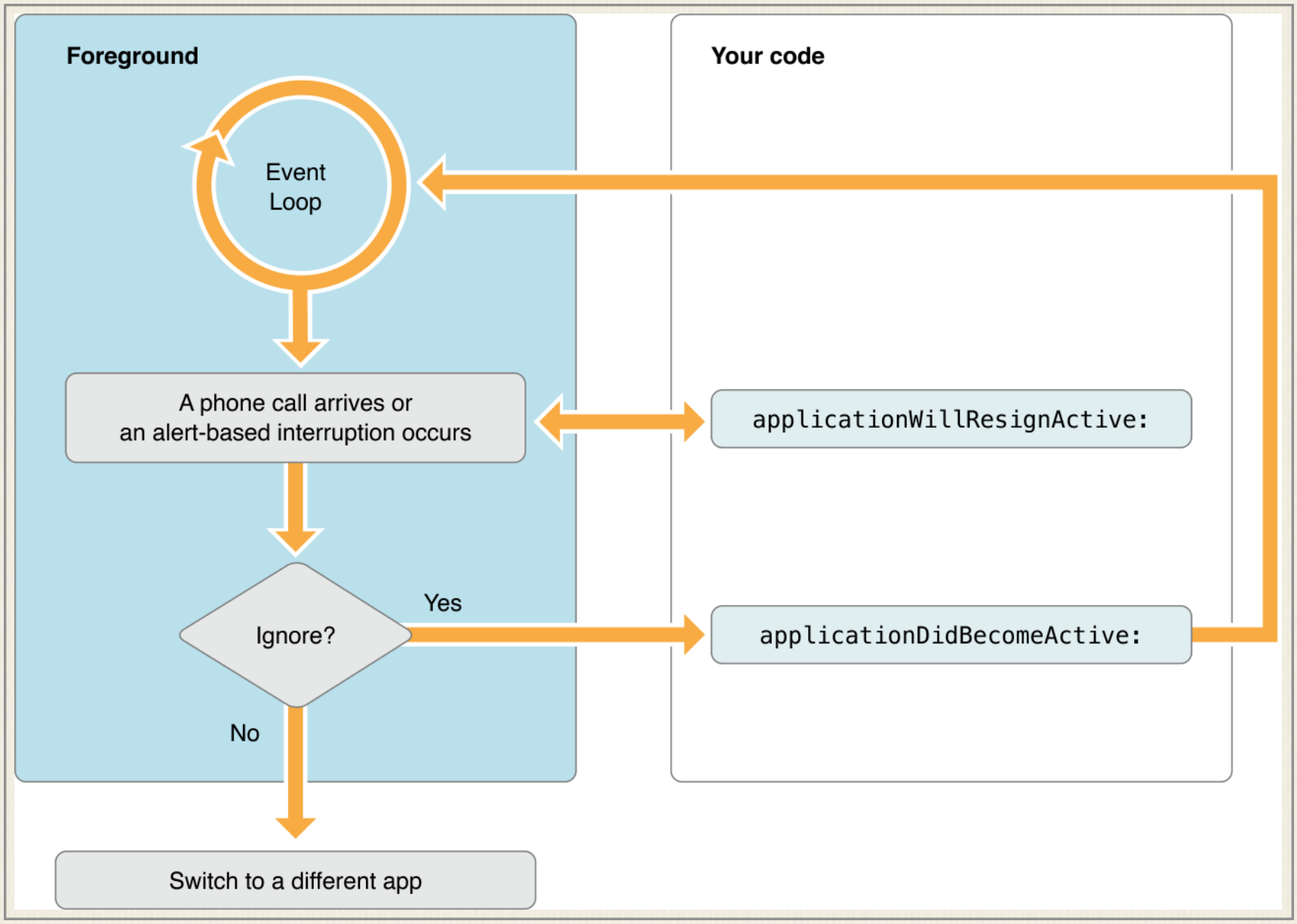
3：加载应用程序进入前台



4：加载应用程序进入后台



5：基于警告式响应中断



当出现这种中断时，我们需要在- (void)applicationWillResignActive:(UIApplication *)application 方法中进行如下操作:

①停止timer 和其他周期性的任务

②停止任何正在运行的请求

③暂停视频的播放

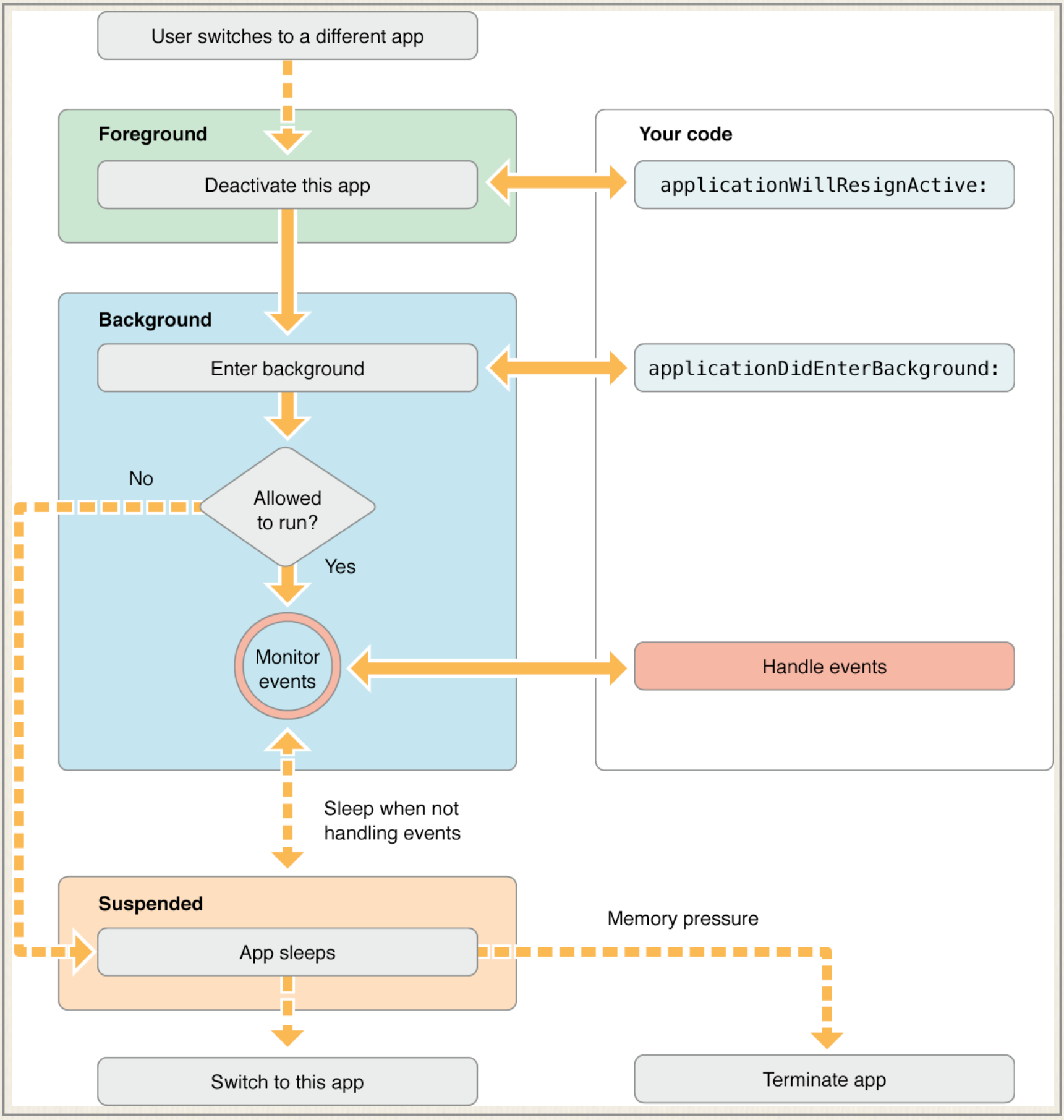
④如果是游戏那就暂停它

⑤减少OpenGL ES的帧率

⑥挂起任何分发的队列和不重要的操作队列（你可以继续处理网络请求或其他时间敏感的后台任务）

当程序回到active状态，我们需要在-(void)applicationDidBecomeActive:(UIApplication *)application 方法中重新开始上述任务。不过游戏要回到暂停状态，不能自动开始。

6：进入后台运行



当应用程序进入后台时，我们应该做些什么?

保存用户数据或状态信息，所有没写到磁盘的文件或信息，在进入后台时，最后都写到磁盘去，因为程序可能在后台被杀死。

释放尽可能释放的内存。

- (void)applicationDidEnterBackground:(UIApplication *)application方法有大概5秒的时间让你完成这些任务。如果超过时间还有未完成的任务，你的程序就会被终止而且从内存中清除。

如果还需要长时间的运行任务，可以在该方法中调用

```
1 [application beginBackgroundTaskWithExpirationHandler:^{
2
3         NSLog(@"begin Background Task With Expiration Handler");
4
5     }];
```

应用程序在后台时的内存使用：请求后台运行时间和启动线程来运行长时间运行的任务。

在后台时，每个应用程序都应该释放最大的内存。系统努力的保持更多的应用程序在后台同时 运行。不过当内存不足时，会终止一些挂起的程序来回收内存，那些内存最大的程序首先被终止。

事实上，应用程序应该的对象如果不再使用了，那就应该尽快的去掉强引用，这样编译器可以回收这些内存。如果你想缓存一些对象提升程序的性能，你可以在进入后台时，把这些对象去掉强引用。

下面这样的对象应该尽快的去掉强引用:

①图片对象

②你可以重新加载的 大的视频或数据文件

③任何没用而且可以轻易创建的对象

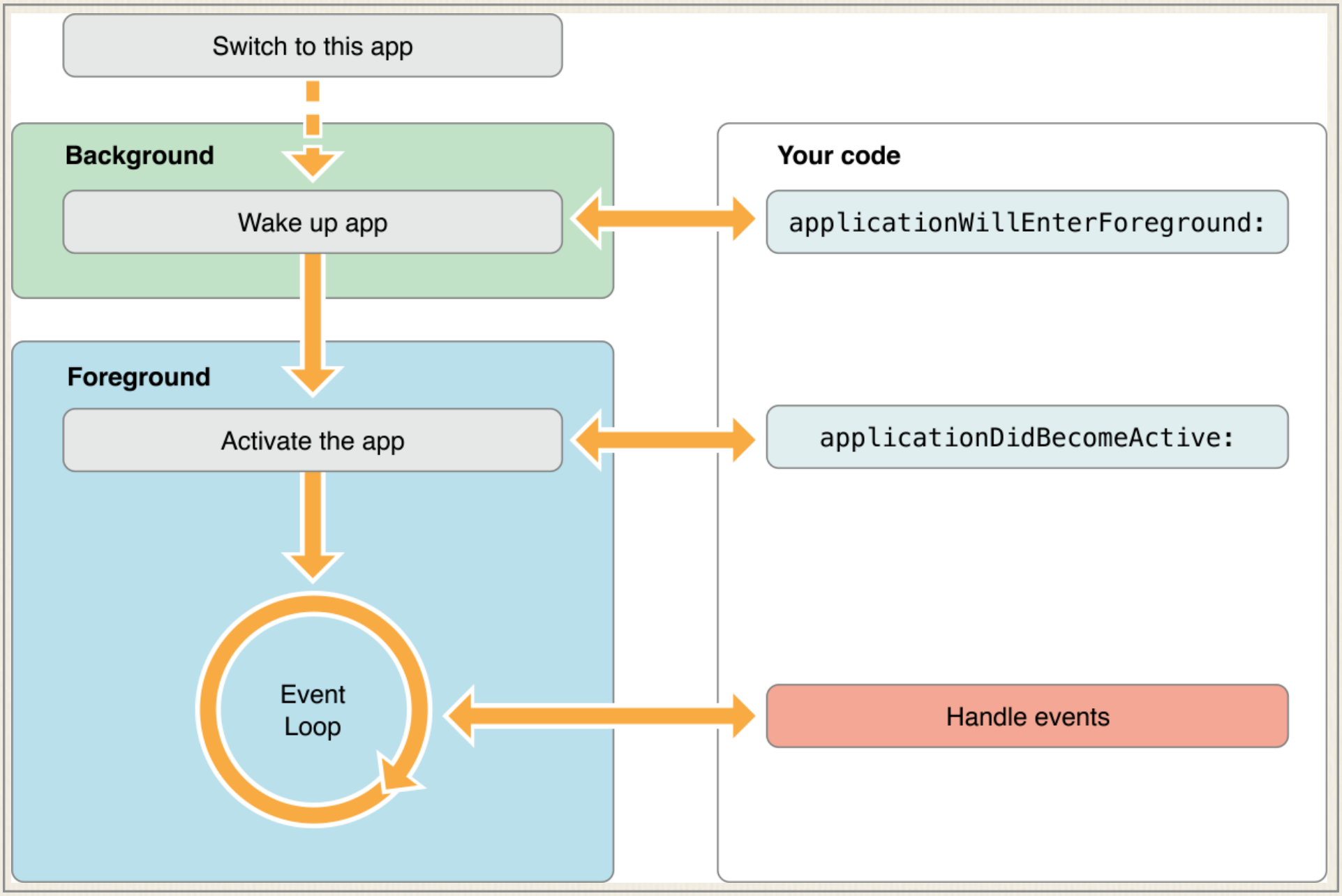
在后台时，为了减少程序占用的内存，系统会自动在回收一些系统帮助你开辟的内存。比如:

①系统回收Core Animation的后备存储。

②去掉任何系统引用的缓存图片

③去掉系统管理数据缓存强引用

7：返回前台运行



在暂停状态的应用程序必须准备处理任何排队的通知时，它返回到前台或后台执行状 态。暂停的应用程序不执行任何代码，因此不能处理与方向的变化，时间的变化，偏好的变化，以及许多其他会影响应用程序的外观或状态的通知。为了确保这些更 改不会丢失，系统排队许多相关的通知，并把它们传递给应用程序，只要它开始再次执行代码（无论是在前景或背景）。为了防止由偏快转为超载与它恢复时通知您 的应用程序，该系统凝

聚事件，并提供一个单一的通知（每个相关类型），反映了净变化，因为你的应用程序**被暂停**。

8：程序终止

程序只要符合以下情况之一，只要进入后台或挂起状态就会终止：

①iOS4.0以前的系统

②app是基于iOS4.0之前系统开发的。

③设备不支持多任务

④在Info.plist文件中，程序包含了 UIApplicationExitsOnSuspend 键。

app如果终止了，系统会调用app的代理的方法 -(void)applicationWillTerminate:(UIApplication *)application，这样可以让你可以做一些清理工作。你可以保存一些数据或app的状态。这个方法也有5秒钟的限制。超时后方法会返回程序从内 存中清除。

注意：用户可以手工关闭应用程序。

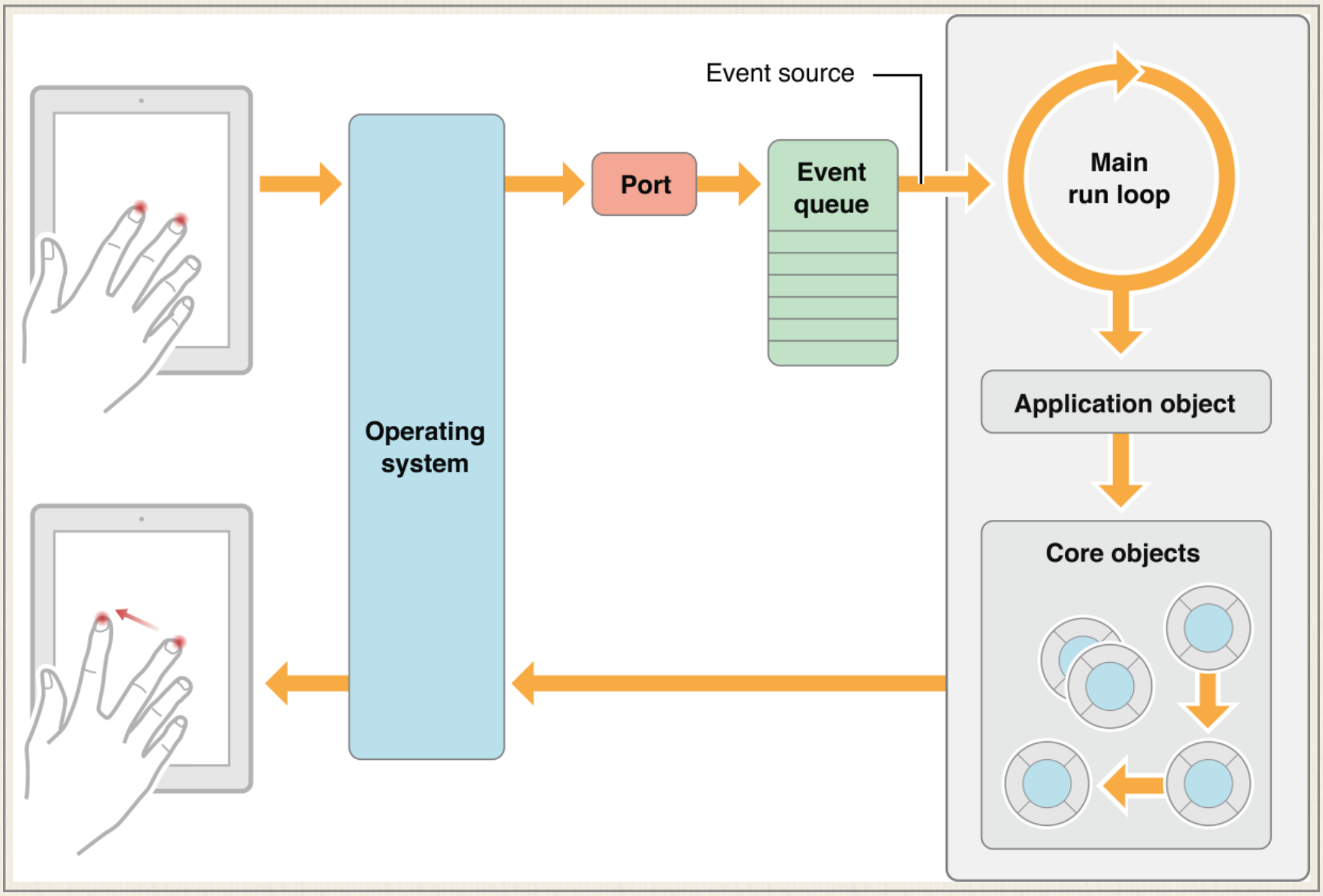
9：The Main Run Loop 主运行循环

Main Run Loop负责处理用户相关的事件。UIApplication 对象在程序启动时启动main run Loop，它处理事件和更新视图的界面。看Main Run Loop就知道，它是运行在程序的主线程上的。这样保证了接收到用户相关操作的事件是按顺序处理的。

用户操作设备，相关的操作事件被系统生成并通过UIKit的指定端口分发。事件在 内部排成队列，一个个的分发到Main run loop 去做处理。UIApplication对象是第一个接收到时间的对象，它决定事件如何被处理。触摸事件分发到主窗口，窗口再分发到对应出发触摸事件的 View。其他的事件通过其他途径分发给其他对象变量做处理。

大部分的事件可以在你的应用里分发，类似于触摸事件，远程操控事件（线控耳机等） 都是由app的 responder objects 对象处理的。Responder objects 在你的app里到处都是，比如：UIApplication 对象，view对象，view controller 对象，都是resopnder objects 。大部分事件的目标都指定了

resopnder object，不过事件也可以传递给其他对象。比如，如果view对象不处理事件，可以传给父类view或者view controller。



本文根据官方文档整理，翻译基本靠谷歌。

原文链接：http://www.cnblogs.com/EverNight/p/3818570.html

英文原文链接：https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/ManagingYourApplicationsFlow/ManagingYourApplicationsFlow.html#//apple_ref/doc/uid/TP40007072-CH4-SW20

LinkedIn技术高管Jay Kreps：Lambda架构剖析

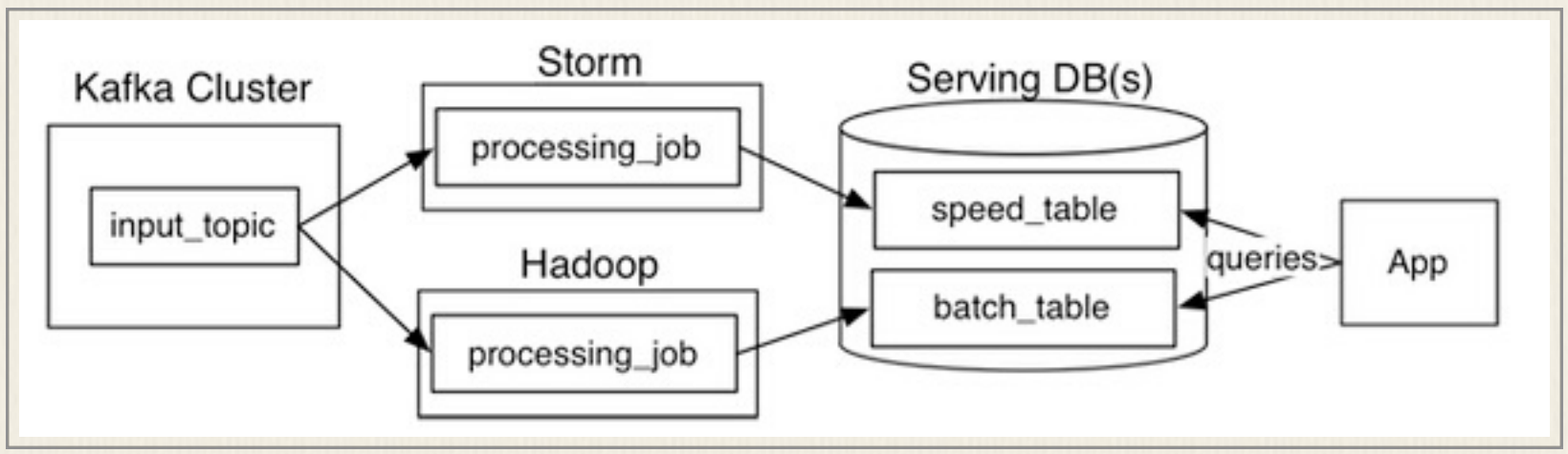
译者：伍昆

摘要：Jay Kreps是LinkedIn的一名在线数据架构技术高管，在日常工作中，Jay Kreps经常被问及有关Lambda架构的问题，为此他结合实际经验和个人体会，针对Lambda架构进行深度剖析，分析了它的优缺点以及采用的替代方案。

Jay Kreps是LinkedIn的一名在线数据架构技术高管，其负责LinkedIn开源项目，包括Apache Kafka、Apache Samza、Voldemort以及Azkaban等项目。在日常工作中，Jay Kreps经常被问及有关Lambda架构的问题，为此他结合实际经验和个人体会，把使用Lambda架构的心得总结为以下几点，我们一起来看下：

Lambda架构的组成

该架构的组成是这样的：



在该架构中，被读取的数据是不可变的，在并行处理过程中数据会依次进入批处理系统（batch system）与流处理系统。从逻辑上看，传输过程发生了两次，一次是在批处理中，一次是在流处理中。在查询时，当这两者都返回结果后，才算是完成一次完整的查询。

这个架构可以有很多的组合变化，我的想法是希望它变得更简洁。例如可以把里面的Kafka、Storm、Hadoop等换成其它类似的系统；惯常的做法是使用两个数据库来存储数据输出表，一个用于实时优化查询，另外一个用于批量优化处理。

Lambda架构的目的是为应用程序提供一个低延迟的复合异步数据传输环境，例如新闻类应用，经常需要进行大规模信息处理，包括输入，归类，索引，存储等操作。

我的体会是：架构的引入不能照本宣科，要根据实际情况进行调整优化。

优缺点分析

优点：

1. **数据的不可变性**。里面给出的数据传输模型是在初始化阶段对数据进行实例化，这样的做法是能获益良多的。能够使得大量的MapReduce工作变得有迹可循，从而便于在不同阶段进行独立调试。
2. **强调了数据的重新计算问题**。在流处理中重新计算是个主要挑战，但是经常被忽视。比方说，某工作流的数据输出是由输入决定的，那么一旦代码发生改动，我们将不得不重新计算来检视变更的效度。什么情况下代码会改动呢？例如需求发生变更，计算字段需要调整或者程序发出错误，需要进行调试。

此外，外界对于Lambda的评论还有其它观点，例如说实时数据处理是固有的，较批处理是高损耗低效率，我对此并不赞同。诚然流处理目的框架前没有MapReduce那样成熟，但是我们应该用发展的观点来看待而不是就此盖棺定论。

缺点：

1. 代码的维护需要在两个复杂的分布式系统中进行，这是个困难的工作。例如说在基于分布式框架上如Storm或Hadoop中进行编程，使用不同的框架就需要进 行不同的处理。虽然目前有解决方案尝试从实时和批处理

框架中进行抽象综合，从而使得既能在高阶框架中进行编程而后又能在流处理或MapReduce中完成 降阶处理操作；Summingbird采用的就是这种方法。不过这样虽然在某程度上能减少编码的复杂度，但是仍然没有根本性地解决问题。

2. 即使能解决二次编码问题，分别在两个系统中运行和维护应用程序仍然是个艰巨的任务。所以，基于不同系统而抽象得到的新系统，都只能是从两者的功能交集中来进行。类似地，在跨数据库系统中进行ORM(对象关系映射)也是个困难的工作。

完成的试验

在LinkedIn 中，我们做了不少的试验。尝试建立不同类型的分布式计算框架，甚至是开发出使代码能在实时或Hadoop中无缝运行的专用API。不过，目前来看还不没做 到完美无缺。因为多系统的编程任务实在是太艰巨了。此外，API的使用会让系统的漏洞不易被发现，同时对开发者有更严格的技术要求——对API的运用要足 够的熟练，从而能在调试或效率检视时能够对所有影响的因素做到了然于胸。

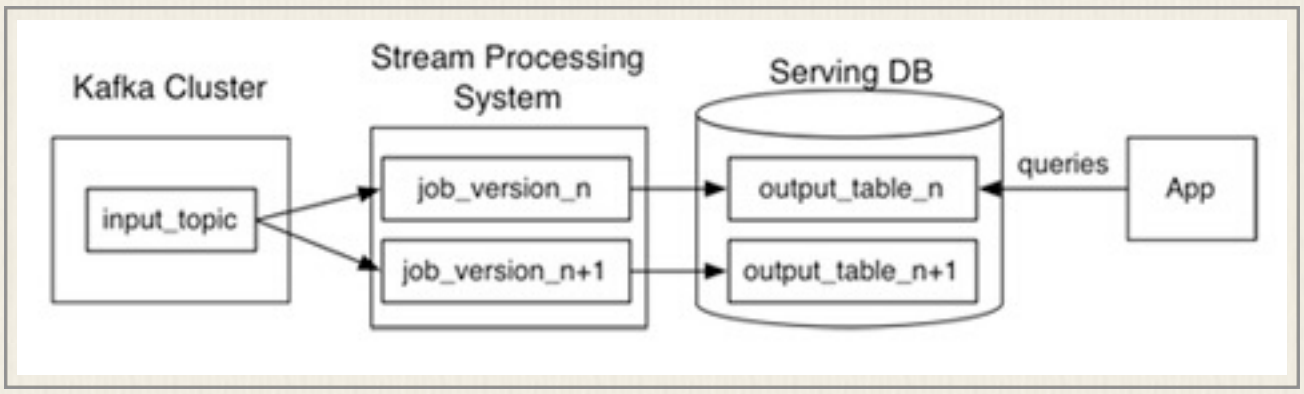
我的建议是：如果你不太看重延迟问题，可以尝试使用类似MapReduce的批处理框架或者是流处理框架，但是最好不要同时使用，除非真的有必要。

那么，Lambda架构的优势是什么呢？我想是它能够很好地指导如何搭建一个复杂的低延迟处理系统。例如搭建一个处理历史数据的高延迟大数据处理系统和一个低延迟的流处理系统来减少重新计算的问题。但是这应该是暂时性的，未来还会存在更好的替代解决方案。

一个替代方案

很多时候，惯性思维会让人觉得流处理系统在处理历史数据等大数据场合不太适合，但是我觉得这可能与他们使用的系统自身限制有关。流处理其实是在数据输出的中间阶段进行的，完成后再把结果返回给用户，所以是具备大数据处理能力的。

例如，请看下我们的一个替代方案：



1. 使用Kafka或其它系统来对需要重新计算的数据进行日志记录，以及提供给多个订阅者使用。例如需要重新计算30天内的数据，我们可以在Kafka中设置30天的数据保留值。

2. 当需要进行重新计算时，启动流处理作业的第二个实例对之前获得的数据进行处理，之后直接把结果数据放入新的数据输出表中。

3. 当作业完成时，让应用程序直接读取新的数据记录表。

4. 停止历史作业，删除旧的数据输出表。

有别于Lambda架构，上述方法是在代码改动时才进行数据重新计算。如果想加快处理速度，计算作业的并行处理能力是个突破口。或许我们可以把这个架构称呼为Kappa架构，虽然它真的很简易。

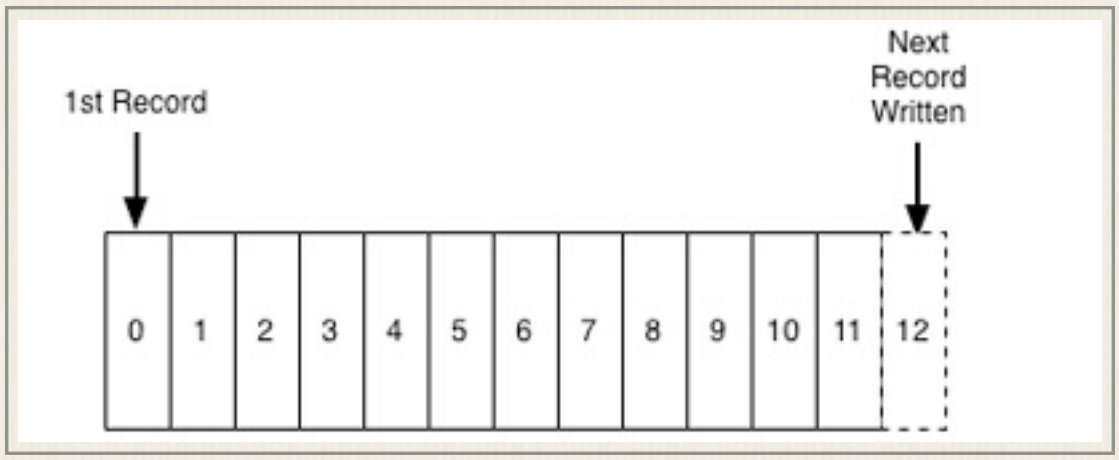
这个方案还能继续优化改进。很多情况下，我们可以把两个数据输出表整合在一起。这样一来，我们可以很快地在程序中读取历史记录和进行其它重要的针对新老版本的测调试工作。

请注意，这个方法不是说让我们远离HDFS，而是说我们不在HDFS中进行重新计算。详细的说明文档，请点击这里进行查阅。

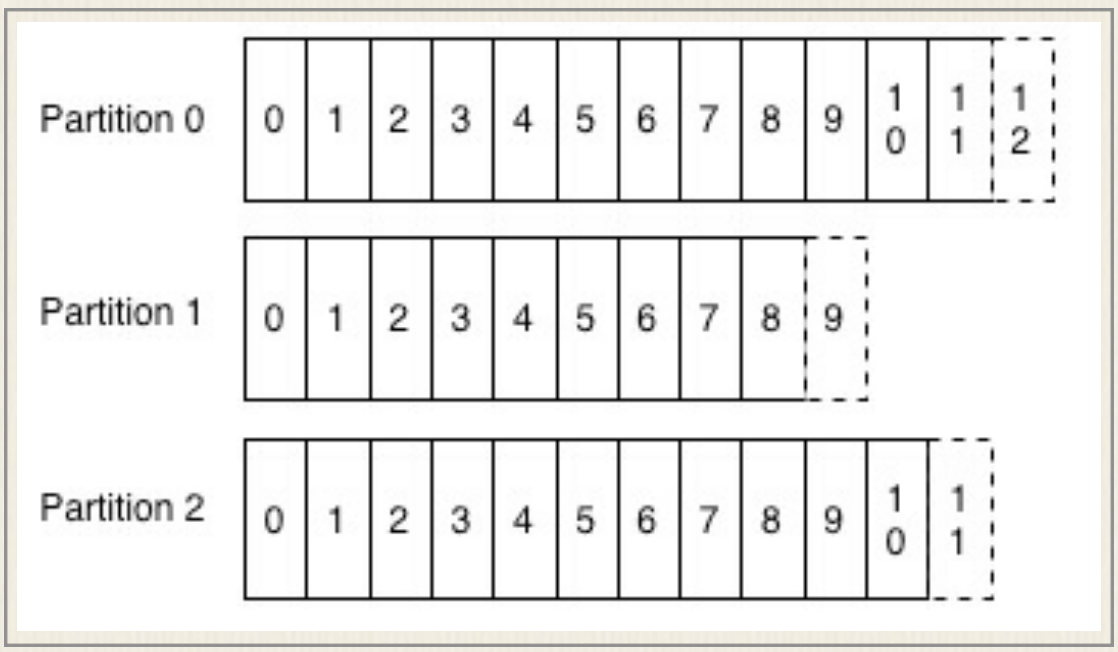
背景知识

对Kafka不太熟悉的读者，可能理解上会有点困难。我们这里给出一些背景介绍，希望对初学者有所帮助。

Kafka中是这样按序来进行日志记录的：



一个Kafka“主题”包含了以下的记录集：



一个流处理在使用这些数据时进行的是“偏移”量维护，就是把每个分区最新加入数据的编号进行记录。所以，只需使用不同的偏移量来重新执行作业就可以实现重新计算的目的。对统一数据添加第二个消息消费者其实就相当于使另外一个读者指向不同的日志位置。

Kafka提供了复制和高容错的能力，使得能在便宜的商业硬件中使用，特别是在TB级别的存储场合中。

写在最后

虽然在LinkedIn中，配合该方法的是Samza流处理系统，但并不代表不能在其他系统中运用。有心的读者可以进行尝试，我很乐意看到有新的方案产生。

原文链接：http://www.csdn.net/article/2014-07-08/2820562-Lambda-LinkedIn

英文原文链接：http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html

全球最杰出的14位程序员

译者：伍昆

摘要：ITWorld整理全球最杰出的14位程序员，包括Jon Skeet、Linus Torvalds、Jeff Dean等。一起来看下让我们膜拜的大神都有哪些?

近日，ITWorld整理全球最杰出的14位程序员，一起来看下让我们膜拜的这些大神都有哪些？（排名不分先后）

1. Jon Skeet

个人名望：程序技术问答网站Stack Overflow 总排名第一的大神，每月的问答量保持在425个左右。

个人简介/主要荣誉：谷歌软件工程师，代表作有《深入理解C#(C# In Depth)》。

网络上对Jon Skeet的评价：

- “他根本不需要调试器，只要他盯一下代码，错误之处自会原形毕露。”
- “如果他的代码没有通过编译的时候，编译器就会道歉。”
- “他根本不需要什么编程规范，他的代码就是编程规范。”

2. Gennady Korotkevich

个人声望：编程大赛神童

个人简介/主要荣誉：年 仅11岁时便参加国际信息学奥林比克竞赛，创造了最年轻选手的记录。在2007-2012年间，总共取得6枚奥赛金牌；2013年美国计算机协会编程比赛 冠军队成员；2014年Facebook黑客杯冠军得主。截止目前，稳居俄编程网站Codeforces声望第一的宝座，在TopCoder算法竞赛中暂 列榜眼位置。

网络上对Gennady Korotkevich的评价：

- “一个编程神童。”
- “他太令人惊讶了，他相当于我在白俄罗斯建立了一支强大的编程队伍”
- “彻底的编程天才”

3.Linus Torvalds

个人名望：**Linux之父**

个人简介/主要荣誉：

- Linux和Git之父，一个开源的操作系统；
- 1998年EFF(电子前沿基金会)先锋奖得主；
- 2000年英国计算机学会Lovelace奖章得主；
- 2012年千禧技术奖得主；
- 2014年IEEE(电气和电子工程师协会)计算机学会先锋奖得主；
- 2008年入选计算机历史博物馆名人堂；
- 2012年入选互联网名人堂。

网络上对Linus Torvalds的评价：“他简直优秀得无与伦比。”

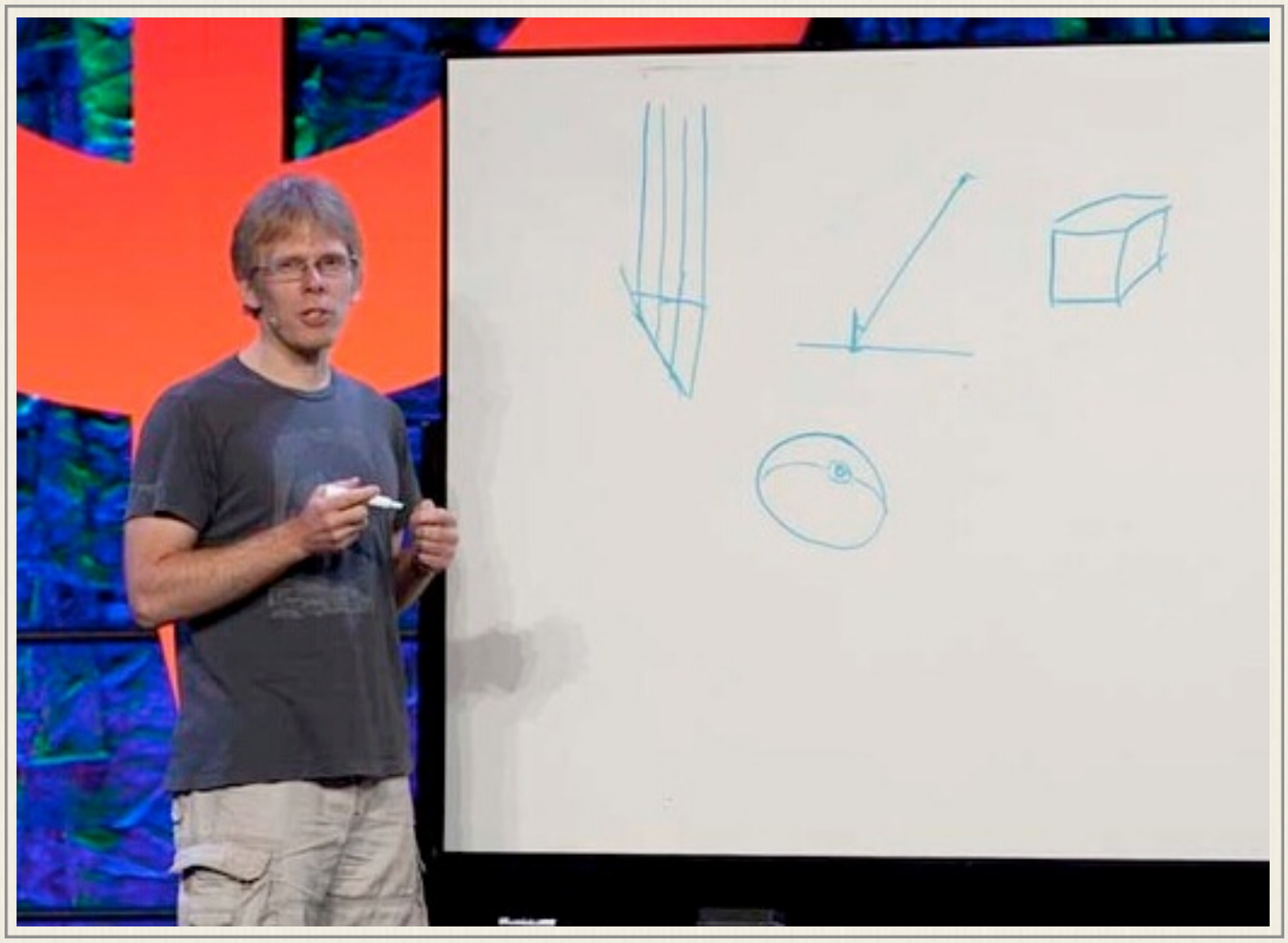
4. Jeff Dean

个人名望：谷歌搜索索引技术的幕后大脑。

个人简介/主要荣誉：谷 歌大规模分布式计算系统的设计师，例如：站点爬行，索引与搜索，在线广告，MapReduce，BigTable以及Spanner(分布式数据库)。 2009年进入美国国家工程院；2012年美国计算机协会SIGOPS Mark Weiser Award以及Infosys Foundation Award奖项得主。

网络上对Jeff Dean的评价：

- “使数据挖掘取得了突破性发展。”
- “能够在各项工作都已安排得满满的情况下，仍能构思、创作、发布出MapReduce以及BigTable这些令人赞叹不已的工具。”



5.John Carmack

个人名望：第一人称射击游戏经典师祖《Doom》(毁灭战士)之父

个人简介/主要荣誉：id Software公司联合创始人，制作了很多脍炙人口的游戏，如：《德军司令部》（Wolfenstein 3D，又名《刺杀希特勒》）、《Doom》(毁灭战士)、《Quake》(雷神之锤)。引领了很多计算机显示领域的新技术，包括：adaptive tile refresh(切片适配更新)、binary space partitioning(二元空间分割)、surface caching(平面缓存)；2001年进入互动艺术与科学学院名人堂；2010年收获游戏开发者精选奖终身成就奖殊荣。

网络上对John Carmack的评价：

- “制作了很多革命性的第一人称射击游戏，影响了一代又一代的游戏设计者。”
- “他能在一周内就完成任何的基础设计工作。”
- “他是会编程的莫扎特。”

6. Richard Stallman

个人名望：Emacs文本编辑器，多种语言编译器GCC的创造者。

个人简介/
主要荣誉：GNU项目发起人，开发出很多核心工具，例如：Emacs，GCC,GDB和GU Make Free Software公司创始人。1990年获得美国计算机协会Grace Murray Hopper奖项；1998年获得EFF(电子前沿基金会)先锋奖。

网络上对Richard Stallman的评价：

- “曾独自一人与一众Lisp黑客好手进行比赛，那次是Symbolics对阵LMI。”
- “尽管我们对事物有不同看法，但他一定是最有影响力的程序员，无论现在还是将来。”

7. PetrMitrechev

个人名望：最有竞争力的程序员之一。

个人简介/主要荣誉：分 别在2000年与2012年收获国际奥林匹克信息竞赛金牌；2011年与2013年赢得Facebook黑客杯赛；在2006年赢得谷歌Code Jam程序设计大赛以及TopCoder算法公开赛；截止目前，暂列TopCoderPetr算法竞赛首位，在Codeforces中排行第五。

网络上对PetrMitrechev的评价：

“即使在印度，他都是程序设计竞赛者心中的偶像。”

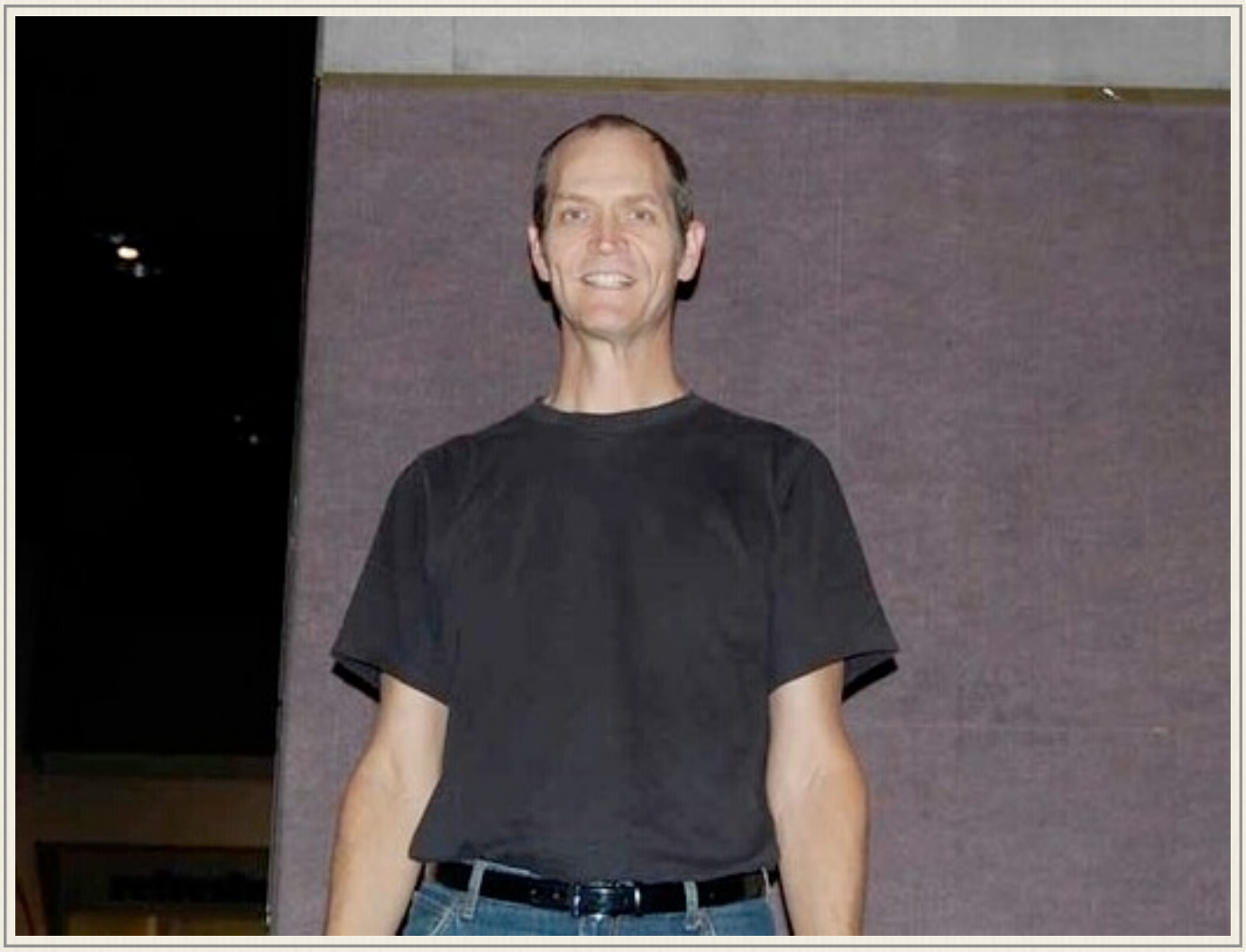
8. FabriceBellard

个人名望：开发出模拟处理器的自由软件QEMU。

个人简介/主要荣誉：开 发了许多著名的开源软件，例如：QEMU硬件模拟虚拟平台，FFmpeg多媒体数据处理软件，Tiny C编译器，LZEXE解压缩软件。在2000年与2001年赢得国际C语言混乱代码设计大赛冠军；2011年赢得谷歌O’Reilly开源设计奖；前圆周 率计算精度世界纪录保持者。

网络上对FabriceBellard的评价：

- “他的作品总是令人印象深刻和光芒四射。”
- “世界上最有创造力的程序员。”
- “他是软件工程领域的尼古拉·特斯拉。”



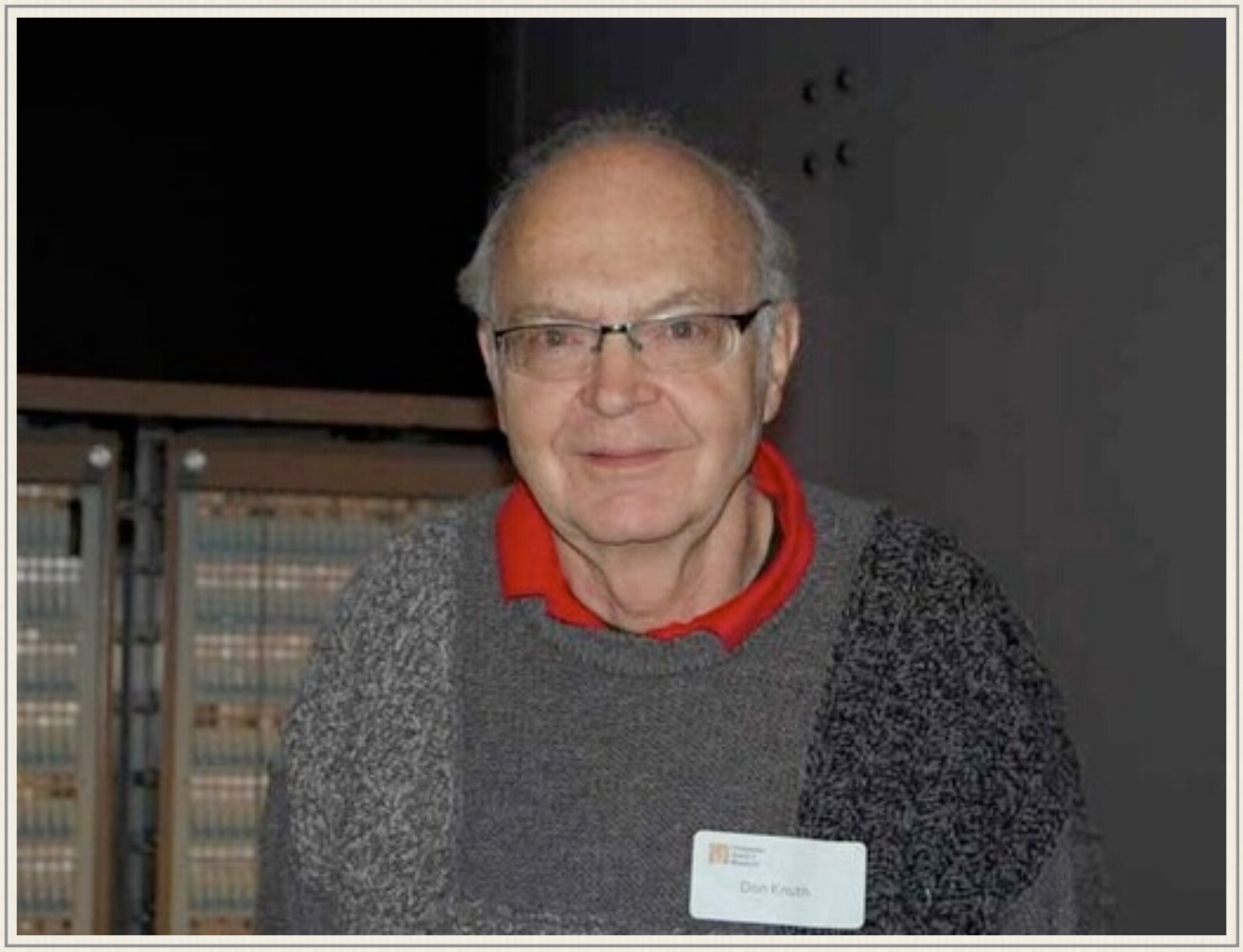
9. Doug Cutting

个人名望：开发出开源全文检索引擎工具包Lucene。

个人简介/主要荣誉：除了Lucene，还开发了著名的网络爬虫工具Nutch，分布式系统基础架构Hadoop，这些大师级作品都是开源的。目前任职Apache软件基金会主席。

网络上对Doug Cutting的评价：

- “他开发出卓越超群的全文检索引擎工具包(Lucene/Solr)以及为世界打开了一扇通往大数据的大门。”
- “开源的Lucene以及Hadoop为全球创造了无数的财富以及就业机会。”



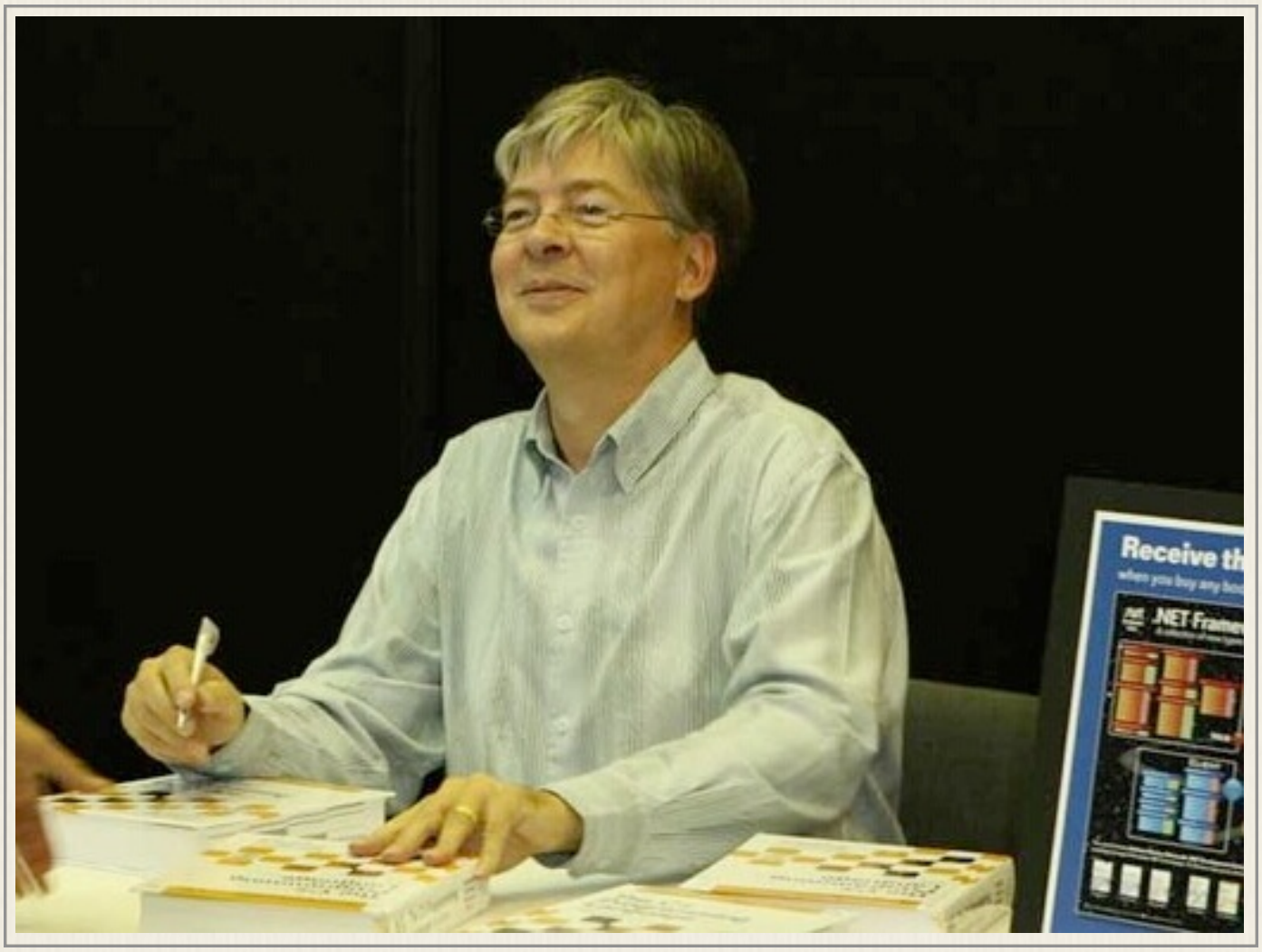
10. Donald Knuth

个人名望：《计算机程序设计艺术》(The Art of Computer Programming)一书的作者。

个人简介/主要荣誉：著 有数本影响深远的程序设计理论书籍；发明了TeX数字排版系统；在1971年成为首位获得美国计算机协会Grace Murray Hopper奖项的人士；1974年获得美国计算机协会A.M. Turning奖项；1979年被授予国家科技奖章；1995年被授予电气和电子工程师协会John von Neumann奖章；1998年入选计算机历史博物馆名人录。

网络上对Donald Knuth的评价：

“我曾经有幸使用过一款无限接近零错误的大型软件，它就是TeX。”



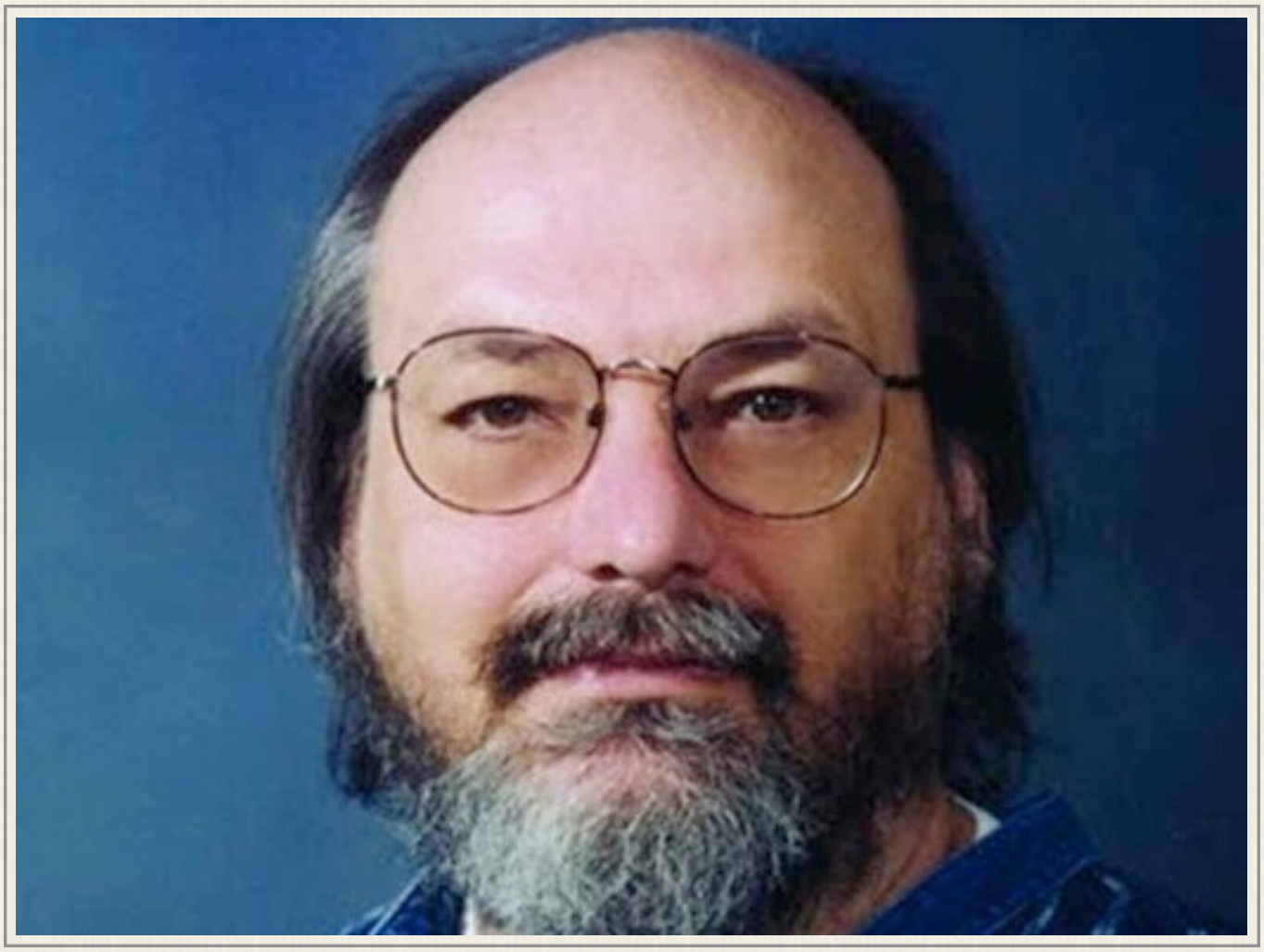
11. Anders Hejlsberg

个人名望：创造了Turbo Pascal。

个人简介/主要荣誉：Turbo Pascal的原作者，Turbo Pascal是最受欢迎的Pascal编译器之一，也首次为Pascal带来整合的开发环境。主导开发了Turbal Pascal继承者Delphi。首席C#设计师与架构师；2011年获得Dr.Dobb’s Excellence in Programming荣誉。

网络上对Anders Hejlsberg的评价：

“我崇敬的程序大师，是我通往专业软件设计师道路上的领路人。”



12. Ken Thompson

个人名望：创造了Unix

个人简介/主要荣誉：与Dennis Ritchie一起创造了Unix。

同时也是B程序语言，UTF-8编码，ed文本编辑器的创造者、设计者。Go程序语言的开发者之一。1983年与Ritchie一起被授予美国计算机协会A.M. Turning奖项；1994年IEEE(电气和电子工程师协会)计算机学会先锋奖得主；1998年被授予国家科技奖章；1997年入选计算机历史博物馆名人录。

网络上对Ken Thompson的评价：

“世界上最杰出的程序员。”

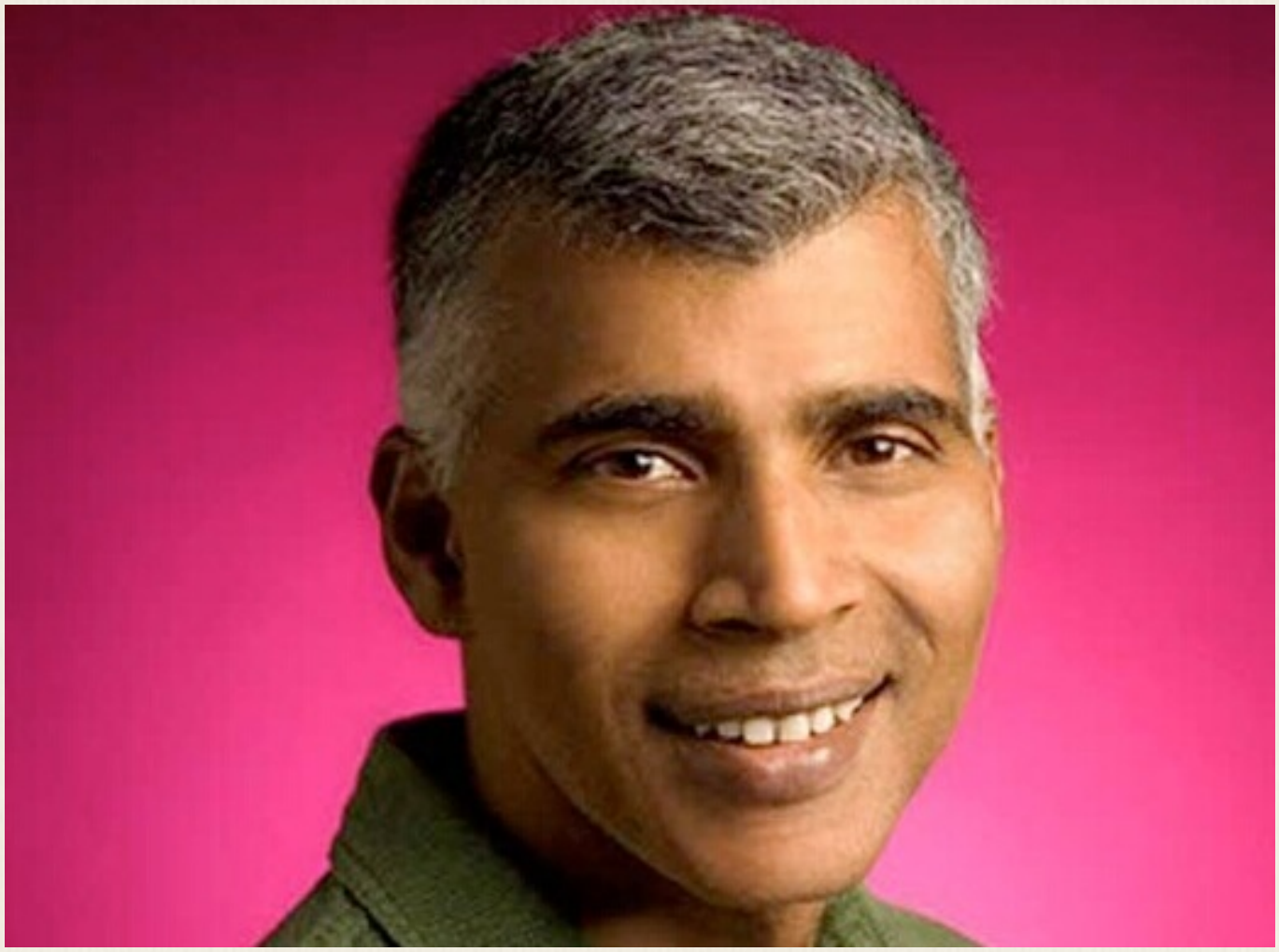
13. Adam D'Angelo

个人名望：问答SNS网站Quora的创办人之一。

个人简介/主要荣誉：前Facebook CTO、研发副总裁，创建了news feed(信息流)的基础架构。SNS网站Quora的创办人之一。2001年以高中生身份参加美国计算机奥林匹克竞赛，最终取得第八名的佳绩。2004 年帮助加州理工学院摘下ACM国际大学生程序设计大赛团体银牌。2005年进入Topcoder大学校际算法竞赛决赛

网络上对Adam D'Angelo的评价：

- “一位程序设计全才。”
- Mark Zuckerberg的评价：
- “我做的每一个好东西，他都能做出六个。”



14. Sanjay Ghemawat

个人名望：Google架构师团队中的核心人物。

个人简介/主要荣誉：帮 助Google设计并推出了大型发布式计算系统，包括：MapReduce、BigTable、Spanner以及Google文件系统。开发出Unix ical日历系统；2009年进入国家工程院；2012年美国计算机协会Infosys Foundation Award奖项得主。

网络上对Sanjay Ghemawat的评价：

“Jeff Dean的最佳拍档。”

原文链接：http://www.csdn.net/article/2014-07-11/2820615-14-world-best-programmers

英文原文链接：http://www.itworld.com/slideshow/158256/superclass-14-world-s-best-living-programmers-425412#slide1

Linus，一生只为寻找欢笑

作者：池建强

摘要：每个人桌面上一台电脑，这曾经是无数计算机先驱的梦想，这个梦想很早就实 现了，在1997年，乔老师和比老师就说过，「比尔，我们共同控制了100%的桌面系统市场」，当然乔老师没说的是，比老师控制了97%，乔老师还不到 3%。时至今日，乔老师走了，比老师颓了，移动终端把传统的 PC 市场冲击的七零八落。普通用户都知道了Windows、Android、OS X 、iOS、BlackBerry等等，但是，他们依然不了解的是另一款在计算机发展史上起到了革命性作用的操作系统：Linux！

每个人桌面上一台电脑，这曾经是无数计算机先驱的梦想，这个梦想很早就实现了，在1997年，乔老师和比老师就说过，「比尔，我们共同控制了100%的桌面系统市场」，当然乔老师没说的是，比老师控制了97%，乔老师还不到3%。时至今日，乔老师走了，比老师颓了，移动 终端把传统的 PC 市场冲击的七零八落。普通用户都知道了Windows、Android、OS X 、iOS、BlackBerry等等，但是，他们依然不了解的是另一款在计算机发展史上起到了革命性作用的操作系统：Linux！

当大家使用 Google 搜索时，使用 Kindle 阅读时，使用淘宝购物时，使用 QQ 聊天时，很多人并不知道，支撑这些软件和服务的，是后台成千上万台 Linux 服务器，它们时时刻刻都在进行着忙碌的运算和数据处理，确保数据信息在人、软件和硬件之间安全的流淌。可以这么说，世界上大部分软件和服务都运行在 Linux 操作系统之上，神马云计算、大数据、移动互联网，说起来风起云涌，其实没有 Linux 全得趴窝（微软除外）。

但是，Linux 和它的缔造者 Linus Torvalds 一样低调，这么牛逼的一个物件，居然只有程序员知道它的传奇，这不科学！所以我准备在这个系列中写写Linus Torvalds：他是 Linux 和 Git 的缔造者，他是一个传统的黑客，与

沃兹一样，少年成名，崇尚自由，一生只为寻找欢笑，他，是一个真正的程序员。

注：为了防止大家把 Linux 和 Linus 搞混，我在后面的文章中统一采用 Linus 的中文译名：李纳斯。

李纳斯在2001年出过一本自传，叫做《Just for Fun》，是他和大卫·戴蒙合著的，当年我有幸读到这本书，了解了很多李纳斯的生平轶事，那时我就琢磨，这个天才已经达到人生的巅峰了吧，结果这位兄台并 未停止前进的步伐，转手就在2005年搞出了分布式版本控制系统 Git，目前几乎全世界的程序员都在用 Git管理他们的代码，著名网站 Github 就是基于 Git 构建的。无论是 Linux 还是 Git，得一即可得天下，结果这哥们以一己之力发起了俩项目，而且都是主力开发人员。最终的结果是，成全了程序员，陶冶了用户，造福了一方百姓。正如李纳斯自己所言：「My name is Linus, and I am your God.」

Linus（一）——生命的意义

1969年末，李纳斯出生于芬兰的赫尔辛基市，算是赶上了60后的尾巴。小时候他是个其貌不扬的孩子， 除了一个鼻子长的「富丽堂皇」之外乏善可陈。他为了让鼻子看上去小一些，经常戴上眼镜就不愿意摘下来，这个策略和现在的很多大脸女生购买三星的 Galaxy Note 手机有异曲同工之妙。幼时的李纳斯不修边幅邋里邋遢，不怎么费劲数学和物理就学得极好，社交圈却一塌糊涂，他母亲经常和别人说，这孩子非常好养，只要把他 放到一个有电脑的小黑屋里，然后再往里扔点薯条和意大利面，就行了。李纳斯对此表示认同。

李纳斯把年幼的自己定位成 Nerd（书呆子），但是从他的自传里我却感受到了这位天才的有趣之处。他在书的前言里写到：

我对生命的意义有种理论。我们可以在第一章里对读者解释生命的意义何在，这样就可以吸引住他们。一旦他们被吸引，并且付钱买了书，剩下的章节里我们就可以胡扯了。（注：做人要厚道啊）

关 于生命的意义，李纳斯的解释是，有三件事具有生命的意义。它们是你生活当中所有事情的动机。第一是生存，第二是社会秩序，第三是娱乐。生活中所有的事情都 是按这个顺序发展的，娱乐之后便一无所有。因此，

从某种意义上来说，生活的意义就是要达到第三个阶段。你一旦达到了第三个阶段，就算成功了。但首先要越过 前两个阶段。

为什么李纳斯会这么说呢，我摘段原文给大家看看，非常有趣：

李纳斯：我给你举个例子来说明这一观点。最明显的是性，它开始只是一种 延续生命的手段，后来变成了一种社会性的行为，比如你要结婚才能得到性。再后来，它成了一种娱乐。大卫：性为什么是娱乐?李纳斯：好吧，我是在对牛弹琴。 我举一个别的例子。大卫：别别，还是说说性吧李纳斯：它是在另一个层次上的blablabla……大卫(自言自语)：哦，参与就是娱乐，而不是在一旁观 看。好，我明白了。

那生存、社会秩序和娱乐又是如何与技术扯上关系的呢?

Linus（二）——天才也疯狂

李纳斯是这么解释的，技术的诞生同样是为了人类的生存，而且是为了让人生活的更好。汽车让人跑的更快， 飞机让人飞得更高，互联网让人懂得更多，手机让人通信更快，一旦这些技术成了规模，就要并入社会秩序，然后下一个阶段就是娱乐，别看手机现在就是个打电话 的工具，但是很快会进入娱乐阶段……（12年后的今天，手机已经彻头彻尾变成了一个娱乐工具，打电话反而成了附属功能）。

李纳斯说：「一切事物都将从生存走向娱乐，但这并不意味着在某个局部地区没有倒退的现象，而且毫无疑问许多地方都有这种情况。有时事物的发展往往分裂的。」

从这些内容我们可以看出，李纳斯有自己的一套理论，而且能自圆其说，其实每个人都有自己的理论，一件事 做或者不做，都是自己说服自己，每一次进步，要么是推翻自己的理论，要么是完善自己的理论。李纳斯在很小的时候就建立了自己的理论领地，那就是数学、物 理、逻辑，最后是计算机，所以他绝不是自己描述的 Nerd，而是一个大智若愚的牛娃，就像射雕里的郭靖一样，看着傻，其实比谁都精，脑子里装的都是十年二十年后的事儿。而且李纳斯比郭靖牛的地方是，就一 个启蒙老师，还是自己的外公，和郭靖一比，高下立判！李纳斯基本上就是个自学成才的典范。

李纳斯的外公是赫尔辛基大学的一位统计学教授，数学家。他有一台Commodore VIC-20计算机（Commodore 是与苹果公司同时期的个人电脑

公司，曾经创造过一系列辉煌，1994年破产），这台电脑的主要功能就是没有功能，你唯一能做的事情就是用 Basic 语言在上面编写自己的程序，老爷子当年就是这么做的，比如做一些数学运算和公式计算等。但是老爷子年老眼花，也不愿意打字，于是就把自己的外孙李纳斯放在 腿上，让他帮助录入写在纸片上的程序。这种很有场面感的场景一再出现后，李纳斯除了对数学有了初步的认识，同时也把计算机玩的娴熟，很快他就在外公的指导下开始编写自己的程序。

评：很多大师级的人物，很小的时候就能在某个领域内头角峥嵘，展现出一 些东西，然后经过长期的练习和创作，最终成为一代传奇。在这个过程里，环境是很重要的，逆境出人才基本上是个伪命题，这句话唯一的作用就是遇到困难时给自 己打打鸡血。李纳斯就是个高知子弟，10岁人家就开始玩计算机了，我们10岁在干什么，打沙包么？甩方宝么？即使你在计算机方面有出众的天赋，但18岁以 前连计算机的面儿都没见过，你就只能默默的牛逼了。等你真正开始展现出自己才华的时候，人家操作系统已经开发出来了，一入世就差别人十年的身位，除了冷冷 的绝望，你还能感受到什么？

所以现在人们没事就北上广深杭，不是喜欢人多嘴杂空气差，而是在这些一线城市可以接触更多的人和事物，见更高的山，渡更宽 的河。不是为了情怀，而是拥有格局。见都没见过，还同一个起跑线呢，一跑就得趴窝。所以，无论这些地方环境多恶劣，竞争多激烈，来的永远多过走的，不为别的，只是为了缓解些许绝望的感觉……

李纳斯用外公的计算机学会了 Basic 语言，并开始编写各种简单有趣的游戏，然后他又发现了 Basic 并不是计算机唯一能理解的语言，在它的下面，还有一种语言是由0和1组成，可以直接被计算机识别，于是李纳斯又开始用机器码编程，这次他可以控制更多计算 机的细节，他与机器变得更加亲密。然后李纳斯就开始上中学了，中学的几年于他而言，其实没有太大变化，因为那些年他几乎都是坐在电脑前面度过的，在这个阶 段，他熟练的掌握了汇编语言。

终于有一天，李纳斯向编程世界挺进的步伐变得缓慢下来，因为他上大学了，原因之一是他必须集中精力读书，原因之二是找不到什么项目去做。还有一件事，李纳斯开始服兵役了，那段时光对他来说是如此特殊：

在手执武器上了一个月的「体育课」之后，我便觉得在我有生之年完全有资格从此一动不动，享受平静的生活了。惟一可做的事情就是把编码打入键盘，或者手里端着一瓶比尔森啤酒！

Linus（三）——改变一生的书籍

终于，让李纳斯痛苦不堪的兵役终于结束了，除了敲锣打鼓欢庆重生之外，他开始继续拓展自己的编程之路， 这时候，生命中最重要的一本书出现了，书的名字叫做《操作系统：设计和实现》，作者是 Andrew S. Tanenbaum。用李纳斯的原话表述就是「这本书把我推上了生命的高峰」。

那个时代 Unix 已经开发出来了。最早 Unix 是用汇编写的，开发过程中 Unix 的两位创始人Ken Thompson 和 Dennis Ritchie觉得用汇编写程序实在是太苦逼了，男人应该对自己好一点！于是老哥俩决定用高级语言来完成下一个版本，他们首先尝试了Fortran，失 败！然后又基于BCPL（Basic Combined Programming Lanugage）创建了 B 语言，B 语言可以被认为是那个时代的解释型语言，不能直接生成机器码，效率上完全没法满足系统的需求，再次失败！我们都知道，一再失败的情况下总会有一位英雄人物 挺身而出，这次是 Dennis Ritchie，他从失败的大坑中爬起来拍拍土抹抹泪，继续对 B 语言进行改造。这次 Dennis 为 B 增加了数据类型，并让 B 语言能够直接编译为机器码，然后又为这门语言起了个极其响亮的名字：「New B」，读一读神清气爽，念一念气冲云霄，从此一代语言巨星冉冉升起，40 年后依然排在兵器排名榜第一位，怎一个牛字了得！当然，Dennis可能考虑了十 几年后中国人民的感受，把「New B」改为了 C 语言，并用 C 语言重新编写了 Unix 的内核，Unix 与 C 从此珠联璧合，长相思守，再也无法分离。

操作系统、Unix 和 C 语言可以说是李纳斯心目中神山上的三座圣杯，为了至高无上的荣耀，他首先要攀上峰顶，把这三座圣杯捧在手中，然后再琢磨建造自己的宫殿的事儿。在那一年的 夏天，李纳斯开始了高强度的阅读和学习，用他的话说就是做了两件事，「一件事是什么都没做，另一件事是读完了719页的《操作系统：设计和执行》。那本红 色的简装本教科书差不多等于睡在了我的床上」。

李纳斯认为，Unix 是一个简洁、干净的操作系统，在 Unix 上的大部分任务都是通过一些基本操作完成的，这些操作被成为「System Call」，顾

名思义，这些操作就是你对系统的呼叫，系统通过响应你的呼叫完成工作。比如 fork、clone（创建子进程），比如 open、close、read、write（文件访问）。这些基本的系统调用通过组合可以完成大部分功能。同时，Unix 还提供了极为强大的IPC（进程间通信）方式：pipe（管道）。很多工作在 GUI（图形界面）软件环境下的读者，最常用的IPC操作可能是复制、粘贴、 鼠标拖拽，这些操作虽然简单，但是必须由人来完成，想要自动化就很困难。而这些在 Unix 上实现起来就像大自然一样自然，你只需要在程序之间开辟出一段缓冲区作为管道，然后父进程和子进程就可以通过这个管道实现进程间通信了。举个例子，以前给 大家介绍的查找历史命令的脚本，就利用了管道的功能，如下：

```
history | grep apache
```

这行命令的含义就是查找包含 apache 的历史命令，其中特殊字符【 | 】用来告诉命令行解释器（Shell）将前一个命令的输出通过「管道」作为接下来的一行命令的输入，就这样，一个简单的进程间通信就完成了。

总之，李纳斯在读完这本书之后，就像郭靖修习了九阴真经全本一样，对机器和代码的世界有了更为透彻的认知，接下来的事情就是等待一个打造传奇的机会。

等待的过程中，李纳斯也没闲着，他又开始编程了。好的程序员对编程的喜爱是溢于言表的，以下摘录一些李纳斯的编程感想：

对于喜爱编程的人来说，编程是世界上最有趣的事，比下棋有趣得多！因为你可以自己制订游戏规则，而你制定什么样的规则，也就会随之出现与此规则相符合的结果。

在电脑世界中，你就是创世者，你对所发生的一切拥有最终的控制。如果你功力深厚，你可以是上帝——在一个较小的层面上。

你可以建筑一个这样的房子，有一个活板门，既稳固又实用。但是每个人都可以看出一个仅仅以坚固实用为目的的树上小屋和一个 巧妙地利用树本身特点的美妙小屋之间的差异。这是一个将艺术和工程融为一体的工作。编程与造树上小屋有相似之外……在编程中，实用的考虑往往被置于有意思、美观优雅或有震撼力的考虑之后。

在代码的世界里，李纳斯就是一个诗人！

Linus（四）——Linux 诞生

Unix 始于上个世纪60年代，在70年代得到了迅猛的发展，这时候的李纳斯还躺在祖父公寓里的摇篮里睡大觉，如果不是后来 Unix 王国自乱阵脚，出现阵营分裂和法律纠纷，可能 Linux 系统根本都不会出现。真实的情况是，Unix 浪费了大把的时间和机会，似乎就是为了等待这个大鼻子、头发纷乱的芬兰小子长大，然后一决高下。李纳斯赢得了自己的时间，他一刻不停的磨练自己的技艺，在 清晨的微光中练习算法，在赫尔辛基的雪山上编译代码，随时随地补充的粮草和武器。二十一年之后，李纳斯抚着雪亮的刀锋上路了，他要去追寻属于程序员的最高 荣耀。

1991年一月，李纳斯花费了三千五百美元，分期付款购买了一台杂牌组装电脑，内存4兆，CPU 33兆赫，还有一台14英寸的显示器，然后又买了 MINIX 操作系统，用十六张软盘把这个操作系统装到了计算机里。之后，李纳斯又用了一个月的时间，了解了 MINIX 的好和不好，并把这个系统改装成了自己得心应手的「战斗机」，开始了战斗的人生。就是在这台电脑上，催生了 Linux 的初始版本。

Linux 的诞生离不开 MINIX，MINIX 是 Mini Unix 的缩写，是 Andy Tanenbaum 教授编写的迷你版的 Unix 操作系统，源代码可以提供给大学和学生，用于操作系统教学，采用了微内核设计。其中的代码还作为《操作系统：设计与实现》的示例程序，这本书我们在 Linus（三）中提到过，给了李纳斯极大的启发。

李纳斯使用了 MINIX 之后，发现这个系统有很多缺陷，比如性能问题、内核问题、文件系统问题，最大的问题是终端仿真器，也就是我之前总提到的 Terminal，登录学校里的 Unix Server 和上网时，李纳斯都需要终端，但是 MINIX 无法满足这个需求。如果普通人遇到这种问题，估计就是发会呆然后洗洗睡了，或者说「你行你上啊」，李纳斯不是普通人！

他决定抛开 MINIX，从硬件层面开始，重新设计一个终端仿真器。牛人就是不同凡响，这个决定表明了李纳斯需要从 BIOS、CPU等硬件层面重新开发出一套系统，除此之外，还需要了解如何把信息写入显示器，如何读取键盘输入，如何读写调制解调器，早期储备的汇编语言 和 C 语言能力终于派上了用场……

两个月之后，终端仿真器完成，对此李纳斯非常骄傲：

对于我了不起的成就，萨拉（妹妹）是了解的。我把终端显示给她看，她盯 着显示器看了大约五秒钟，看着上面是一串A和一串B，说了声「很好」，然后就没有然后了。我意识到我的成就并不辉煌，这犹如你指给人看你铺设的一条长长的 柏油马路，但想向别人解释这条马路的意义是完全不可能的。

当时是三月，也可能是四月，就算彼得盖坦街上的白雪已经化成了雪泥我也不知道，当然我也并不关心。大部分时间我都穿着睡衣趴在相貌平平的计算机前面噼噼啪啪的敲打键盘，窗户上的窗帘遮得严严实实，把阳光和外部世界与我隔离开来。

Linux 操作系统就这样开始了，一发不可收拾。李纳斯的当时编程状态是这样的：编程——睡觉——编程——睡觉——编程——吃饭——编程——睡觉——编程——洗澡——编程……

实现了终端仿真器之后，李纳斯马不停蹄，开始添加磁盘驱动和文件系统，那一年李纳斯还在上课，但是课程 很简单，他唯一的课外活动就是参加每周三晚的同学聚会，这个长着大鼻子的技术天才，常常会因为担心自己缺乏社交能力和容貌丑陋而失眠，对那时的他来说，唯 一有趣的事情就是把驱动程序写出来。于是他咬咬牙对自己说，还得干下去。（看来没有女神的好处就是可以写个操作系统出来，然后把自己叫做 上帝）。

随着工作的进展，终端仿真器正在开始向一个操作系统的方向发展，李纳斯显然也看清楚了这一点。

在整个创造 Linux 的过程中，我们没有看到李纳斯使用了什么样高级工具，估计那时也没有，整个系统基本上是一行行代码敲出来的，纯手工打造，这些先贤的编程功底和效率让我们 叹为观止，所以，现在，我决定打开终端，输入 vi，然后键入：to be continued，感受一下李纳斯当年编程的风采……

随着李纳斯不断的敲击键盘，他的终端仿真程序也不停的扩张，从刚开始的小树苗长成了一株盘根错节的大 树，树根牢牢的抓住土地，枝丫努力的伸向天空，花朵和果实开始在高远的天空中烁烁发光，所有的细节都在李纳斯的掌控之中。懂行的技术人员都看得出来，这个 大鼻子的芬兰小子是准备开发一个操作系统啊。

是年6月份，李纳斯基本确定了要开发一个操作系统内核的计划，并开始着手搜集 Unix 操作系统标准的相关资料。1991年7月3日，格林威治时间上午10点钟，李纳斯在 MINIX 新闻组发出了一封求助邮件，寻求有关 Posix 标准的帮助，他在邮件中写道：

目前我正在 MINIX 系统下做一个项目，对 POSIX 标准很感兴趣。有谁能向我提供一个机器可读的最新的 POSIX 规则？如果能有个 FTP 地址就更好了。

这份公开的邮件是标识 Linux 问世的最早证据。邮件发出后不久，有人就寄来了厚厚的 POSIX 标准，同时赫尔辛基工学院的 Ari Lemke 也对李纳斯的邮件做出了响应，为李纳斯提供了一个 FTP 地址，用来上传他即将完成的操作系统。

注：POSIX 全称是可移植操作系统接口（Portable Operating System Interface）。IEEE 最初制定 POSIX 标准，是为了提高 UNIX 环境下应用程序的可移植性。随着技术的发展，POSIX 开始不局限于 UNIX 系统，后续的 Linux 和 Windows NT 都部分的遵循了该标准。POSIX 在李纳斯开发的过程中起到了灯塔的作用，直接后果就是 Linux 系统从一开始就走在了正规军的康庄大道上，基本没有跑偏过。Linux 几乎可以适配各种类型的硬件体系结构。

标 准和 FTP 地址都有了眉目，李纳斯开始实现各种 System Call，以便让 Shell 运行起来。这段时间的工作让李纳斯时常感到灰心丧气，看着增加的代码量，工作似乎前进了一大步，但是检验一下功能又仿佛没有任何进展。有时候他还不得不放 弃之前的想法和已经完成的代码实现，另辟蹊径重头再来，即使是在天才面前，代码也能让人欢喜让人忧。

终于 Shell 已经可以在新的操作系统上工作了，李纳斯开始编写拷贝（cp）和列表（ls）等程序。Shell 程序一旦完成，就好像完成了从0到1的飞跃，一切都变得无比顺利，李纳斯面前仿佛出现了一条阳关大道，一切都豁然开朗了，他说，要有光，于是就有了光。对 于这种状态，李纳斯表示：「我很满意」，并且开始用「Linux」称呼这个操作系统。

这种满意非常重要，因为那个夏天李纳斯除了伏在电脑面前噼噼啪啪的敲击键盘，什么都没做。芬兰四月到八 月的日子是一年中最美好的时光。人们到布满小岛的海上航行，去海滩上晒日光浴，到夏日小木屋中消遣时

光。但是李纳斯，他只是在永无休止的编写程序，忘记了 白天和黑夜，黑色的窗帘遮蔽了灿烂的阳光，也遮蔽了外面的世界。他唯一的想法就是，得赶紧把这该死的系统做出来！

1991年8月25日，李纳斯在MINIX 新闻组上发邮件做了一个调查，想知道大家希望这个新的操作系统具备什么特征。

1991年9月17日，李纳斯把已经完成的新操作系统上传到了Ari Lemke提供的 FTP 服务器上，并准备用「Freax」作为操作系统的最终代号，结果遭到了Ari Lemke的激烈反对。Ari Lemke对李纳斯说：

「李哥，您咋会想到用这么变态的名字命名操作系统呢？原来的Linux不挺好的嘛」「那样不会显得自恋 么？」「您这样就不对了，操作系统是开天辟地的大事，人民群众都等着用您的名字命名呢，看看他们的眼神，您能辜负他们的期望吗？Linux 天生不就是用来了 Unix 遥相呼应的么？这是命，得认！」「这……那我就不推辞了啊」

以上为意译，不过基本上和古代皇帝的黄袍加身是一个意思。新的操作系统最终以「Linux」命名，并在10年后名扬天下，20年后统治服务器领域，可谓 Linux 恒久远，Linus 永流传。

Linux 内核0.01版本终于发布了，虽然漫长的开发过程才刚刚开始，但李纳斯终于可以松口气了：

瞧，我真的做出了点什么。我没有在骗你们。这就是我所做的……

创造操作系统，就是去创造一个所有应用程序赖以生存的基础环境——从根本上来说，就是在制定规则：什么可以接受，什么可以做，什么不可以做。事实上，所有的程序都是在制定规则，只不过操作系统是在制定最根本的规则。——李纳斯

Linus（五）——继续前行

Linux 从一诞生就被打上了开源的烙印，这一点对 Linux 的后续发展起到了至关重要的作用。从1991年内核0.01版本发布，到1994年1.0版本闪亮登场，世界各地无数的开发者为 Linux 提交了代码，李纳斯为 Linux 建立了讨论组 comp.os.linux，全世界爱好开源和 Linux 的程序员与黑客都在上面讨

论问题，他们就像群蜂筑巢一样，不断的通过个体和群体的力量交替推进 Linux 的飞速发展。

李纳斯对自己说：嗯，没有任何东西可以阻挡 Linux 的普及！

这种感觉估计很多程序员都体会过，当你设计的算法得出了正确结果的时候，当你自以为解决了一个海森堡 bug（Heisenbug，表示不可重现）的时候，当你完成了一段精妙代码的时候，你摘下厚重的眼镜，推开铺满灰尘的书桌，打开办公室唯一的窗户，迎着 夕阳把一只废弃的圆珠笔扔出窗外，然后冲着天空大喊：还有谁~~~？这是一种拔剑四顾心茫然的情怀。

李纳斯还不止于此。他不仅单枪匹马写出了 Linux 的内核，而且做出了开源的决定。他把 Linux 放到了互联网上，并且允许那些希望使用和改进它的人们根据开源协议修改和提交源代码。这两点对互联网的影响是极其深远的，估计李纳斯当年也没有想到，当时 的两个小小的涟漪，经过时间和空间的放大，十几年后形成了一股互联网巨浪，到现在 Linux 依然处于风口浪尖。

对于 Linux 取得的成功，李纳斯将其归结为是由自己的缺点导致的：

1、 我很懒散2、 我喜欢授权给其他人

其 实这两个所谓的缺点，正是优秀程序员和领导者必备的要素，它们让 Linux 成为世界上最大的开源协作项目，为喜爱 Linux 的人们带来了最美好的技术和应用，现代的互联网几乎是运行在 Linux 之上的，可以说，李纳斯改变了世界，你每一次伐开心后在淘宝上买包包，都有李纳斯贡献的力量！

Linus（六）——来到硅谷

1996年的春天，Linux 顺利发布了2.0版本。是年李纳斯27岁，这个芬兰小子已经慢慢厌倦了芬兰平淡无奇的日子和不眠不休的编程生活。对于一个技术天才来说，创造一套新的技术 体系就像艺术家完成一个雕像一样，当一块粗砺的岩石在他的亲手打磨下逐渐显山露水，展现出其完美容颜的时候，后续的修修补补会让这些天才产生倦怠的感觉。 他们需要更快的剑，更高的山和更强大的对手。尤其是期间李纳斯访问过两次美国之后，这种感觉变得愈发不可阻挡了。

说起来美国确实是个神奇的国度，这样一个移民国家中，居住了各种从不同国度不远万里跨海而来的种族，每 个种族无论在基因上还是文化上都具有原来国家的特质，这些特质相互融合与对抗，让这块大陆上的人民更锐意进取，更开放，更自由，他们愿意去追求和接纳美好 的事物，最终一不留神把美国搞成了世界文化的大熔炉，而开放的文化和环境又极大的激发了人们的想象力和创造力，近代和现代的科技成果几乎全部源于美国，要 么是美国人搞的，要么是外国人在美国搞的。所以有时候我们也不用顾影自怜，嘲笑自己没有国产的操作系统和编程语言，因为其他国家也没有，或很少有，芬兰好 不容易出了个天才少年，也没好好珍惜，最终落了个「流落」异国他乡的下场。

李纳斯一到美国就被这块新大陆吸引了，一切都是那么的新鲜和美好，他的感受与你第一次出国后在微信朋友圈发的「天是那么的蓝，云是那么的白」是一样一样的。李纳斯在自传中写道：

我所参观的摩门教堂已有一百五十年的历史，却被照顾的很好，清洗后的教 堂显示出亮丽的白色。要是在欧洲，所有的教堂都显得老旧不堪，像是蒙上了一层岁月的斑痕。看着这洁白亮丽的教堂，我脑海里产生的唯一联想竟然是迪斯尼乐 园。因为它看起来太像是童话故事中的城堡，而不太是一个教堂了。

我记得自己徒步走过了金门大桥。在桥的这头时，我望着对岸的马林海岬，恨不得立刻就到对岸去徜徉在那美丽的群山之间。但等 我真走到那边时，我几乎不愿意再挪动双腿……那时的我绝对想不到，在时隔六年以后的今天，我会坐在海风吹拂的海岬峰顶，一面俯瞰太平洋、旧金山湾、金门大桥和笼罩在雾中的旧金山城，一面对着大卫的录音机讲述着这一切。

从美国回到芬兰之后，李纳斯对自己说，我要去美国。

当李纳斯透露出自己的就业计划之后，马上有多家公司递来橄榄枝，其中包括著名的 Linux 公司 Red Hat。这种感觉是如此美妙，就像你刚刚掏出一支香烟，面前已是千百个打火机舞动。但是李纳斯本着不加入任何一家 Linux 公司的原则，拒绝了 Red Hat，参加了另一个名不见经传的公司的面试，这家公司叫做 Transmeta，中译名「全美达」，你们可以从维基百科上查到这家公司，不过我打赌，知道这家公司的读者不会超过千分之一，这并不是咱们孤陋寡闻，因 为美国人民刚开始也不知道这家公司在干嘛，全美

达官网在1997年中上线，两年半后网站的建设情况是「This web page is not yet here」，又过了很久人们才从内部员工透露出的一点信息得知，这家公司似乎是搞处理器的。这是我所知道的唯一一家保密措施强过苹果的公司，如果不是李纳 斯，这家公司就像是根本没有存在过。

就是这样一家公司，面试了在开源社区名满天下的技术天才、Linux 操作系统的缔造者李纳斯，并且将其招至麾下，一待就是六年。从某种程度上，这六年严重的影响了 Linux 操作系统前行的脚步，因为李纳斯没有足够的时间开发 Linux 了。

虽然根据 Transmeta 与李纳斯的协议，他可以继续从事 Linux 的开发，而且他确实也想这么做，比如白天为 Transmeta 工作，编写 X86 解释程序，晚上继续 Linux 的伟大事业。不过真实的情况是，晚上丫睡着了……

关于加班和睡眠，李纳斯是这么解释的：

很多人都认为加班加点的工作才算真正的工作。我可不这么想。无论是 Transmeta 的工作还是 Linux 的工作，都不是靠牺牲宝贵的睡眠时间换来的。事实上，如果你想听真话，我要说，我更喜欢睡觉。

总之，李纳斯第一次从互联网上消失了，很多悲观的开发者纷纷奔走相告，李纳斯这小子是不是被招安了？丫开始为商业公司干活了，Linux 作为自由软件是不是已经濒临死亡了？每当这时候李纳斯就会出来给大家打打气说，哥还在呢，只不过刚睡醒……

关于李纳斯的这段经历，曾经在硅谷工作过的一位朋友给我提供了如下文字，大意是这样的：

每次想起李纳斯这段经历，我都要感慨万千。第一次得知李纳斯虎落硅谷的 事是在2002年夏天，当地的水星报记者先是把李纳斯大吹一通，然后说他从芬兰老家搬到美国，就职于 Transmeta 已五年有余，但 H1 移民仍然停留在劳工卡初级阶段，六年期满就要打道回府了。

当时这份报纸的读者大概有一半人有 H1 经历，然后这一半人里的一半都知道 Linux 是啥东东，但是从未听说过 Transmeta 是何方神圣，这货居然把一代技术英雄扣在那儿为一个名不见经传的小资本家作苦力，导制全球开源事业停滞不前，真是胆大包天啊！于是很多读者跑到水星报去 说，象

李纳斯这样的天才愿意移民到美国，布什亲自开飞机去接都不为过，怎么可以被移民局压了五年呢balabala……

还 好，李纳斯在2003年离开了这个叫做「全美达」的公司，受聘于开放源代码开发实验室（OSDL : Open Source Development Labs, Inc），重新统领开源世界的各路英豪，全力开发 Linux 内核，Linux 再次焕发出勃勃生机，这一次，它要引领的是互联网的技术浪潮……

Linus（七）——关于财富

李纳斯对待财富的态度就是「视金钱为粪土」，是真的粪土。

那种默然的态度让人感觉非常可怕。当一个人随便动动手挂挂名签个字就能获取上千万美金的时候，他依然和 自己的妻女一家人挤在圣克拉拉一栋两层楼的公寓套房里，过着一个普通程序员的生活，同时不断改进已经遍布全球的 Linux，这是什么精神？这是毫不利己专门利人的国际主义战士的精神。

写到这我不禁想起了绿茵场上的冰王子博格坎普，当他接到几十米外的长传，用标志性的慢速停球过掉扑上来 的后卫，轻扣，过掉另一个后卫，颠球，闪过最后的防守，面对守门员的时候不是大力抽射和仰天长啸，而是把球搓出一道完美的抛物线，球越过门将，缓缓落入网 窝，然后博格坎普，低着头慢慢的走开，留给对手的是优雅与实用并世无双的技艺，和令人绝望的背影！

默然的感觉，懂了撒？

很多程序员创业成功或跟随创业成功之后，自以为功成身退，最早扔掉的就是代码和编译器，然后购豪宅当天使满世界贴旅游照片，你们感受一下，这个境界是完全不可同日而语的。（请勿对号入座，如有误伤，必是友军所为）

事实上李纳斯在拿到第一笔真正的财富之前，一直处于日子紧巴巴的状态。当时另两位带头大哥比尔·盖茨和 史蒂夫·乔布斯早已名满天下家私万贯，同时有大量的技术人员、商人和公司通过 Linux 及其相关技术获取了巨额财富，对此，李纳斯的态度是：「和我有毛关系」，他似乎对一大群才气不高的编程人员能够享受到大笔的财富并不在意。这种情况一直持 续到所有的有识之士都坐不住了：李纳斯，你再也不能这样下去了！

伦敦的一位企业家希望李纳斯在他羽翼未丰的 Linux 公司做个董事会成员，报酬是一千万美金。李纳斯说，不用。企业家惊呆了，当他喃喃自语「卧槽你特么知道一千万美金是啥概念吗」的时候，李纳斯已默默走远。

Red Hat 公司为了感谢李纳斯的卓越贡献，为他提供了一些期权，李纳斯的回复同样是，不用了，我不会给你独家的授权许可的。Red Hat 的人差点疯掉：「李爷期权您就收着吧，我们什么都不要行了吧」「唔这样啊，那就放这吧」，这就是李纳斯！

正是这笔期权让李纳斯收获了第一笔巨额财富，因为 Red Hat 1999年8月11日在纳斯达克上市了。李纳斯先是意识到自己从身无分文突然变成了拥有五十万美元的土豪，然后是一百万，五百万，李纳斯终于变得亢奋起来，原来期权也是钱啊！终于不用再为生计发愁了，对着这个事情，李纳斯的定义是：我真是最幸运的家伙！

事实上李纳斯从来没有想过 Linux 能够获得如此巨大的成功。他只是为了自己方便写了一个操作系统内核并想借此获得一点回报而已，「假如我事先知道了要做到如 Linux 这般成功需要做多少基础和琐碎工作的话，那我肯定会相当沮丧的。这意味着你首先要非常优秀，并且你所做的大部分决定都导致了正确的结果。」

任何理智的人在登山之前凝望着高耸入云的山峰和崎岖艰险的山路时，都会陷于沮丧之中。解决办法就是先迈出第一步再说，然后，但行好事，莫问前程。

Linux 不仅给李纳斯带来了名声和财富，同时给大众带去了巨大的好处。年轻一代中最聪明的程序员和黑客都在使用 Linux 的产品，正是开放的 Linux 给这些天才的程序员带去了巨大的创作热情和喜悦，他们在 Linux 平台上完成了一个又一个杰出的作品，这些技术形成的生产力，对互联网的发展起到了巨大的推动作用，直到今天。

Linus（八）——巨星碰撞

在 Linux 出现之前，桌面操作系统的市场基本上是由比老师和乔老师控制的，虽然乔老师控制的少了一些。Linux 出现之后，桌面操作系统的格局并没有太大变化，但是服务器端市场的变化却是翻天覆地的。原本比尔希望通过 Windows NT 和 Server 系列在服务器领域复制桌面操作系统的辉煌，

从而千秋万载，一统江湖。然而，世界的发展永远是多元的，没人能通过一己之力改变历史发展的多维性，比尔·盖茨 也不行。于是 Linux 出现了，并以星星之火可以燎原之势一举拿下服务器操作系统的半壁江山。

一方是商业公司和封闭的策略，另一方是自由软件和开放的协议，这场战争一开始支持率就是一边倒的，李纳 斯就像对抗风车的堂·吉诃德，但是他自己不仅没有遍体鳞伤，还在没怎么亲自出场的情况下把微软这个软件风车搞得狼狈不堪，这种情况发生在现实生活中绝对是 老百姓喜闻乐见的，李纳斯成了自由软件世界里的英雄和领袖，但也就此与微软结下了世仇，比尔和李纳斯许下了永世不相见的誓言。

有些加盟微软的朋友告诉李纳斯，他们曾见到他的头像被钉在了微软公司的飞镖靶心上。李纳斯对此的评价是：一定是我的大鼻子太好瞄准了。

李纳斯与另一位业界巨头苹果之间就没这么激进了，毕竟 Linux 和 OS X 师出同门，都是从老前辈 Unix 那儿毕业的，坐在一起还能唠唠家常，事实上李纳斯和乔布斯确实有过一次历史性的会面。

李纳斯来到硅谷不久，就收到了一封来自乔老师秘书的邮件，邮件中写到：「听闻小李飞刀光临硅谷，蓬荜生辉，老乔不才，重回苹果，以期振昔日之雄风，如得小李相助，必将如猛虎加之羽翼而翱翔四海，天下可得。期待会面。」（当然是意译）

李纳斯看完之后不明白乔布斯要干什么，只是觉得很厉害的样子。毕竟李纳斯还坐在外公腿上拨弄电脑键盘的时候，苹果的沃兹已经纯手动打造出苹果的第一代个人电脑 Apple I 了。李纳斯决定去见一下儿时的偶像，并了解一下苹果的新操作系统。

两代科技巨星的会面被安排在苹果总部 Infinity Loop，乔布斯带着原 Next 公司技术总监 Avie Tevanian（Mach 之父）接见了李纳斯，双方进行了友好而亲切的会谈，然后会谈的结果和某国常规会谈一样，就是没有结果。

其时乔布斯十年放逐回归苹果，举手投足已是大宗师气势，他对李纳斯说，我大苹果虽然现在看起来有点颓， 不过海盗精神永存，我们已经准备好重新起航了。目前个人电脑领域仍然只有两个玩家：微软和苹果。如果 Linux 和苹果能够珠联璧合，那一切将是最好的安排，所有的开源爱好者都

能够用上优雅与极客并存的 MacLinux 了。然后 Mach 之父 Avie Tevanian 向李纳斯详细介绍了整合 Mach 和 Linux 内核作为 OS X 混合内核的计划，之后庞大的 OS X 体系将构建在 Mach 和 Linux 内核的基础之上。 同时乔老师表示，基于 Mach 和 Linux 的内核系统将采用开源的方式运作，这样全世界的开源爱好者都可以为 Mac 和 Linux 开发程序。

这几乎是一个完美的双赢方案，乔老师都被自己描绘的蓝图打动了，永远年轻，永远他妈的热泪盈眶！谁能拒绝苹果公司和乔布斯如此完美的邀请呢？

李纳斯能！

乔布斯认为自己的扭曲现实力场加上苹果巨大的市场潜力一定会让李纳斯怦然心动，没想到这个芬兰小子在计 算机面前待久了，水米油盐不进，任凭乔布斯口吐莲花，我自巍然不动。首先李纳斯对 Mach 就不感冒，他认为 Mach 几乎犯下了所有的设计错误，它让系统变得复杂而效率低下；其次李纳斯觉得乔布斯可能没意识到，Linux 的潜在用户要比苹果系统多；第三李纳斯乐观的认为，虽然 Linux 的目标不是占领桌面操作系统，但是显然「我们很快就能做到这一点了」。所以李纳斯当时的反应是：

为什么我要关心这些？我为什么要对苹果公司的故事感兴趣？我不觉得苹果公司里有什么有趣的事情。我的目标也不是占领什么桌面操作系统的市场。（嗯，虽然 Linux 马上就要做到这点了，但这从来就不是我的目标）

现 在看来，李纳斯当时对 Linux 在桌面操作系统的前景过于乐观了，虽然他天纵奇才桀骜不驯，但是也无法预测到 OS X 和 iOS 在十年后引领移动开发的浪潮。不过即使知道 OS X 未来的大发展，心高气傲的李纳斯也不会接受苹果的收编，因为 Linux 一直是独立和自由的软件图腾。

无论如何，这次非正式的会谈没有达成任何实质性的效果，但是对后来的 IT 格局产生了巨大的影响。苹果不再关注 Linux，而是转向了 BSD。2001年苹果任命 FreeBSD 的发起人之一，老牌 BSD 黑客 Jordan Hubbard 为 BSD 技术经理，后升为 Unix 技术总监，负责 OS X 操作系统底层核心 Darwin 的研发，最终，Mach 与 BSD 技术整合在一起，形成了混合内核。另外，苹果开始觉得开源项目也不是那么靠谱，后续他们先后研发并开源了优秀的编译器项目 LLVM 和 Clang，一举替换了整条 GCC 编译链，为 OS

X 和 iOS 的性能优化和语言特性提供了巨大的帮助。这也算是苹果对那些牛叉哄哄的开源人士的回击：看，我们也可以做开源，而且比你们做的好。

Linux 则继续在开源、独立、自由的方式下一路狂奔，虽然在桌面操作系统领域的成就乏善可陈，但是在服务器端大放异彩，目前几乎整个互联网都是运行在 Linux 及其衍生产品之上的，可以说没有 Linux，互联网不可能得到如此迅猛的发展。

十年以后，移动互联网时代来临。OS X 上长出了 iOS，Linux 上则诞生了 Android，这两个移动开发领域的双子星都有一个老祖宗，那就是 Unix。一次话不投机的会谈让 OS X 和 Linux 分道扬镳，在十几年后的今天，它们又以一种不同的方式相见了，世界永远都是多元的，可能冥冥中自有天意吧。

Linus（九）——Linus 和 Git

很多人在完成了类似 Linux 这样宏伟的软件产品之后，基本上就止步不前了。但是李纳斯却从未停歇创新的脚步。2003年加入开放源代码开发实验室之后，李纳斯重新全职投入 Linux 内核的研发，并开始酝酿自己的另一个跨时代的产品。

2002年，Linux 内核开发团队开始采用 BitKepper 作为代码版本管理工具。BitKeeper 是一套分布式的版本管理工具，它满足了 Linux 内核开发的技术需求。但是 BitKeeper 只是暂时对 Linux 等开源软件团队免费，并不是自由软件。2005年 BitMover 公司不再免费赞助 Linux 开发团队。对此李纳斯表示非常遗憾，但遗憾之后他并没有自怨自艾伤心落泪，而是愤怒的与其他几个小伙伴花了几个星期完成了一套新的分布式代码管理工具，命 名为 Git。两个月之后，Git 发布了官方版本，并在不同的项目中应用，自由软件社区给予了 Git 广泛的支持。

与 SVN 和 CVS 等软件不同的是，Git 更关注文件的整体性是否有改变，Git 更像一个文件系统，它允许开发者在本地获取各种数据，而不是随时都需要连接服务器。Git 的最大的特点就是离线分布式代码管理，速度飞快，适合管理大型项目，难以置信的非线性分支管理。

2005年 Git 发布之后，技术日臻成熟，很多大公司都开始采用 Git 管理自己的项目代码，2008年2月 Github 公司基于 Git 构建了协作式源代码托

管网站 Github，目前该网站是这个星球上最大的源代码集散地，几乎所有的优秀代码都托管在 Github 上。Git 已经成为程序员使用最多的源代码管理工具！

对于 Git 的成功，李纳斯表示：

Git的设计其实很简单，它有一个稳定而合理的数据结构。事实上，我强烈建议围绕着数据来设计代码，而不是反其道而行之，我觉得这可能就是 Git 如此成功的原因。坏程序员总是担心他们的代码，而优秀的程序员则会担心数据结构和它们之间的关系。

从 Git 诞生到今天已经有9个年头了，Git 始终没有背离其设计的初衷：高性能、简单的设计、非线性高并发分支的支持和完全的分布式。

对于李纳斯来说，Git 现在是他的主要消遣工具之一。他很喜欢在 Git 上编程的感觉，因为再也不用担心锁定问题、安全问题和网络问题，这种感觉真是太美妙了！

我们继续期待李纳斯的第三个伟大的作品！

Linus（十）——生活的意义

李纳斯认为生活意义的全部就在于：生存，社会交往和寻找乐趣。因为我们所做的一切事情，最终似乎都是为了我们自己的乐趣。而进化作为主线始终贯穿其中。

李纳斯对进化的理解是：

「你知道在整个太阳系，人类已知的最复杂的工程是什么吗？——不是Linux，不是Solaris，也不是你的汽车。是你，还有我。想想你和我都是怎么来的——不是什么超复杂的设计，没错，凭运气。除了运气，还有：

通过分享「源代码」实现自由的可用性和授粉机制，生物学家把它称作DNA。

毫不手软的用户环境把我们不好的版本轻易地替换成更好的可执行版本，从而使种群更加优秀（生物学家把这叫做「适者生存」）。

大量的无方向的并行开发（试错法）。

我从未如此严肃过：我们人类永远都无法复制出比我们自身更复杂的个体，而自然选择却不假思索的做到了。不要低估适者生存的 力量。不要错误地认为你可以做出比大量的平行试错反馈环更好的设计，那样就太抬举你的智力水平了。说实话，太阳照常升起，这和任何人的工程技巧或者编程风格都没有关系。

李纳斯一生只为寻找欢笑，但是他却取得了无数的成就和荣誉：

1997年，在芬兰赫尔辛基大学计算机科学系，李纳斯接受了他的硕士学位。两年后，他在斯德哥尔摩大学接受名誉博士学位，并在2000年在他的母校获得了同样的荣誉。

1998年，李纳斯接受了电子前哨基金会先锋奖。

2004年，李纳斯被《时代》杂志选为世界上最有影响力的人之一。

2006年，《时代》杂志欧洲版评选李纳斯为过去60年最有革命性的英雄人物之一。

2012年4月20日，李纳斯被宣布成为两位获奖者之一，和山中伸弥共同获得当年的千禧技术奖。该奖被普遍形容为相当于在技术领域的诺贝尔奖。

2012年4月23日，李纳斯进入互联网协会（Internet Society，I-SOC）的网络名人堂。

李纳斯憎恶分明，经常口不择言，比如他对 C++的评价是：C++是一门糟糕的语言。而且有一群不合格的程序员在使用C++，他们让它变得更糟糕了。他对自己的两个产品命名的解释是：我是个自大的 混蛋，我所有的项目都以我的名字来命名。开始是Linux， 然后是Git（英国俚语，饭桶的意思）。

不过我最喜欢李纳斯说过的一句话是：Talk is cheap, Show me the co-de。他一直用自己的编程人生诠释着这句话。2006年的时候，Linux 内核代码的2%依然是李纳斯完成的，他是代码贡献最多的人之一（是年37岁）。到了2012年，他对内核的贡献主要是合并代码，编程变少了，但是他依然对 是否将新代码并入到 Linux 内核具有最终决定权。

李纳斯用自己精彩的编程人生和对自由软件的热爱演绎了现代社会中一个书呆子的胜利。如果你爱一个人，就 让他去编程吧；如果你恨一个人，

就让他去编程吧。代码让我们欢笑，也让我们忧伤，让我们沉默，也让我们高歌。对于程序员来说，代码是这个世界上最美妙的音 乐，会编程的孩子，都是好孩子!

本文参考资料:

李纳斯自传：http://book.douban.com/subject/1451172/

原文链接：http://chijianqiang.baijia.baidu.com/article/21626